Modern Intelligent Systems: A Graduate Companion

Neural Networks, Fuzzy Logic, and Evolutionary Optimization

Haitham Amar

First edition, 2026

Modern Intelligent Systems: A Graduate Companion
Neural Networks, Fuzzy Logic, and Evolutionary Optimization

First edition, 2026
Published by Haitham Amar
Typeset in LaTeX.

Preface

Over time, through teaching courses that engage artificial intelligence both directly—through machine learning and neural networks—and indirectly, at the level of data collection and system design, a recurring gap became apparent. While the literature offers many excellent treatments of individual techniques, it rarely provides a coherent path that guides students from the foundational notions of intelligence to the most advanced models used today. What is often missing is a unified narrative that connects intuition, mathematical formulation, and practical deployment across the diverse tools that constitute intelligent systems.

Courses in machine learning tend to emphasize neural networks and deep learning; others focus on optimization and operations research, from classical search strategies to genetic algorithms. Each approach is valuable, yet time constraints and disciplinary boundaries often force a narrowing of scope, sacrificing breadth for depth or vice versa. In one course in particular, a broader framework emerged—one that treated these methods not as isolated topics, but as complementary responses to recurring modeling challenges. That framework, though not originally mine, proved invaluable: it allowed students to situate linear regression, neural networks, transformers, fuzzy inference systems, and evolutionary algorithms within a single, coherent perspective on intelligent system design. This book is an attempt to make that perspective explicit and durable.

This book has evolved into a concise graduate companion that blends the original chapter voice of ECE 657 with laboratory-style checklists and reflective prompts. The chapters move from supervised learning foundations to fuzzy logic and evolutionary computing, mirroring the trajectory of the original course material while adding connective tissue so that a reader can revisit the material years later without hunting for missing context.

Origins in ECE 657

In 2019, I was asked to teach ECE 657 at the University of Waterloo. At the time, the course leaned heavily toward soft computing, and fuzzy inference systems had constituted a large portion of the material in earlier offerings. Prof. Karray, who built the course, felt it was time to broaden its scope beyond that single

paradigm, and he was generous enough to let me reshape the arc of the course.

Over the following years, I came to view fuzzy inference systems as one important piece in a larger mosaic rather than the organizing principle. I iterated on the syllabus—moving topics, adding and removing chapters, and tightening mathematical through-lines—toward a narrative that is coherent, broad in coverage, and implementable by an engineer.

As of this writing, the field is in the era of large language models, and this book covers them (see Chapters 13 and 14). But it also emphasizes other ideas and toolkits that may underwrite future breakthroughs in intelligent systems: careful probabilistic thinking and diagnostics, principled optimization, sequence modeling beyond any single architecture, and hybrid reasoning approaches that have repeatedly re-emerged in new forms.

The material has since been rewritten to stand alone as a book. Any offering-specific details (schedules, grading, local policies) now live in Appendix C (especially *Using this book in ECE 657*); readers outside that course can ignore the appendix entirely.

Perspective

This book prioritizes ideas that survived multiple paradigm shifts. It emphasizes principles that remain useful even as architectures and tooling change.

For a practical reader's guide—roadmap, prerequisites, and suggested reading paths—see the final sections of Chapter 1. Notation and reading conventions are collected up front in *Notation and Conventions*.

The result is a self-contained reference for researchers and engineers who want a rigorous but narrative-friendly treatment of neural networks, soft computing, and hybrid reasoning systems.

Acknowledgments

This book grew out of teaching and revising the ECE 657 material over multiple offerings. I am grateful to the students whose questions and feedback repeatedly exposed where explanations were brittle and where the narrative needed better bridges between intuition, math, and practice.

I also thank colleagues who shared perspectives on how these topics fit together as an engineering discipline rather than as isolated techniques. Any remaining errors and omissions are my responsibility.

Contents

List of Figures

List of Tables

Notation and Conventions

Symbol overloads. A small number of symbols are intentionally reused across chapters (for example, $\sigma(\cdot)$ as the sigmoid nonlinearity versus σ as a width/scale parameter). For a one-page index of the most common collisions and the disambiguation rule used in this book, see Appendix D.

Symbol	Description
$\mathbf{x} \in \mathbb{R}^n$	Input vector (features)
$y \in \mathbb{R}$	Regression target (continuous)
$y \in \{0, 1\}$	Binary class label (Bernoulli outcome)
$\hat{y}$	Model prediction
$\mathcal{D}$	Dataset or feasible domain
$\mathcal{L}$	Loss function (objective)
$\sigma(\cdot)$	Sigmoid function $1/(1 + e^{-z})$
$\tanh(\cdot)$	Hyperbolic tangent activation
$\mathrm{ReLU}(z)$	$\max(0, z)$ activation
$\mathbf{W}, \mathbf{b}$	Weights and biases (parameters)
h_t, c_t	Hidden and cell states (RNN/LSTM)
n	Sequence length (tokens)
d_{model}	Model (embedding) width
h	Number of attention heads
d_k, d_v	Per-head key/query and value widths
$\mathbf{Q}, \mathbf{K}, \mathbf{V}$	Query, key, value matrices
$\mathbf{W}_i^Q, \mathbf{W}_i^K, \mathbf{W}_i^V$	Per-head projection matrices
$\mathbf{W}^O$	Output projection after concatenating heads
KV cache	Stored past keys/values for decoding
$\mu_A(x)$	Membership of x in fuzzy set A
T, S	t-norm (AND) and s-norm (OR)
$\mathrm{softmax}(\cdot)$	Normalized exponential mapping
$\|\cdot\|_2$	Euclidean norm
∇	Gradient operator
η	Learning rate
λ	Regularization strength
k	Number of clusters/classes/neighbors (context-dependent)
$\mathbb{E}[\cdot]$	Expectation
$\mathrm{Var}[\cdot]$	Variance
$\mathrm{diag}(\cdot)$	Diagonal matrix formed from a vector
$\odot$	Hadamard (elementwise) product
$\phi(\cdot)$	Feature map; in kernels, $k(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^\top \phi(\mathbf{z})$
$\mathbf{1}, \mathbf{I}$	All-ones vector and identity matrix

This section collects book-wide notation, conventions, and a few reading aids. Symbols may be locally redefined within a chapter when explicitly stated. Where symbols are overloaded (e.g., σ as sigmoid vs. standard deviation), the local meaning is made explicit in context.

Conventions

Throughout the book we follow a consistent notational style:

- Bold lowercase ($\mathbf{x}, \mathbf{w}$) denote vectors; bold uppercase ($\mathbf{W}, \mathbf{X}$) denote matrices; plain roman symbols (x, y, σ) denote scalars.
- Random variables are written in uppercase (X, Y) when needed, with lowercase (x, y) for their realizations.
- Transpose is always indicated by the superscript $\cdot^\top$, as in $\mathbf{W}^\top \mathbf{x}$; we avoid bare T to reduce ambiguity.
- The logistic sigmoid is written as $\sigma(z) = 1/(1 + e^{-z})$; the same letter σ without an argument (e.g., σ^2) denotes a standard deviation. The intended meaning is clear from whether an argument is present. In fuzzy chapters, $\mu_A(x)$ denotes membership rather than a mean; context and arguments disambiguate overloaded symbols.
- Embedding matrices are written in bold ($\mathbf{E}, \mathbf{W}, \mathbf{U}$) and should not be confused with the expectation operator $\mathbb{E}[\cdot]$; when expectations appear, they are always typeset with the blackboard bold $\mathbb{E}$. We use row embeddings by convention: a one-hot row vector $\mathbf{x} \in \{0, 1\}^{1 \times |V|}$ selects a row via $\mathbf{x}\mathbf{E} \in \mathbb{R}^{1 \times d}$.
- Feature maps in kernel methods are written as $\phi(\cdot)$; we reserve $\varphi(\cdot)$ for radial-basis kernels. When probability density functions are needed, we write them as $p(\cdot)$ (or $p_X(\cdot)$ when the variable must be explicit). The corresponding design matrix Φ collects feature-map evaluations on data.

These conventions are occasionally restated in local "Notation note" boxes where multiple meanings could collide (e.g., in chapters on statistics or recurrent networks). As a practical guide: row vectors are written as $1 \times n$ objects, column vectors as $n \times 1$; a *design matrix* collects one data point per row (features in columns), so data matrices are row-major $N \times d$.

Reading Aids

How to read the visuals

- **Legends and icons:** Every figure caption states what each color or line style represents (safe vs. heuristic transformations, training vs. validation curves, etc.).
- **Tables and references:** Table captions mention the chapter(s) they support so you can quickly jump back when a later chapter references them.

Editorial heuristics: four recurring questions

Each chapter has been edited with four recurring questions in mind:

- What is the core scientific idea, and how does it relate to earlier material?
- Which methodological cautions should a practitioner keep close at hand?
- How do the accompanying figures or derivations anchor those ideas visually?
- Where does the topic sit within the broader landscape of intelligent systems?

Author's note: intuition before algebra

The math that follows is intentionally tight, but the spirit of this book is to keep the *reason* for each tool front and center. When I introduce a method, I start from the question an engineer would actually be wrestling with and then introduce the equations needed to make it precise. Read each chapter with that heuristic in mind: first ask what story the technique lets you tell, then check that the derivations honor that story.

Part I: Foundations and the ERM toolbox

1 About This Book

Learning Outcomes

After this chapter, you should be able to:

- Explain what this book means by an *intelligent system* (and how that differs from an *intelligent machine*).
- Place modern AI ideas in a brief historical context (logic, computation, and learning).
- Use the book's organizing lenses (system components, levels of intelligence) to interpret later chapters.
- Navigate the book structure and reading paths using the roadmap figure.

Intelligent systems are engineered artifacts that perceive, reason, and act under constraints. This chapter introduces the working vocabulary used throughout the book—historical context, core definitions, and recurring design themes. The roadmap in Figure 1 anchors the strand sequence and reading paths.

Design motif

We treat "intelligence" operationally: specify what a system represents, what actions it can take, and how it checks itself against objectives and constraints.

1.1 Historical Foundations of Intelligent Systems

A brief historical sketch helps place intelligent systems within a longer tradition that runs from early mechanical devices, through symbolic logic and computation, to modern machine learning.

Mechanical Automata and Scholastic Logic In the 12th–13th centuries, engineers such as Al-Jazari designed programmable water clocks and mechanical

automata whose gears, cams, and valves executed fixed sequences of actions. Although these devices lacked learning or internal models, they embodied the idea that artifacts could sense (via floats and levers), transform signals mechanically, and act on their environment. In parallel, medieval scholars such as Ibn Sīnā and Thomas Aquinas refined Aristotelian syllogistic logic, systematizing patterns of valid inference even though a fully symbolic notation did not yet exist.

The Mechanical Computer and Early Programming In the 19th century, Charles Babbage designed the mechanical computer now known as the *Analytical Engine*. Ada Lovelace is often cited as one of the first programmers; her notes on the Analytical Engine include an algorithm for computing Bernoulli numbers and helped establish programming as a discipline.

An important (and still practical) lesson from this era is the "garbage in, garbage out" principle: if incorrect input is provided to a computational system, the output will also be incorrect. In modern terms, this is a reminder that data quality and validation are part of the intelligence pipeline, not an afterthought.

Mathematical Logic and Formal Reasoning The symbolic formalism used in modern AI emerged in the 19th and early 20th centuries. Works by George Boole (1847), Gottlob Frege (1879), Giuseppe Peano (1889), and later Bertrand Russell and Alfred North Whitehead (1910–1913) introduced algebraic and predicate-calculus notations that underpin automated reasoning. Formal inference rules such as:

$$\text{If } A = B \text{ and } B = C, \text{ then } A = C. \tag{1.1}$$

This exemplifies the transitivity of equality—an example of a valid inference rule operating on equality relations—and provides a basis for reasoning systems that manipulate symbols according to formal rules.

The Turing Test and the Birth of AI The mid-20th century marked a pivotal moment with Alan Turing's proposal of the *Turing Test* in 1950. This test was designed to assess a machine's ability to exhibit intelligent behavior indistinguishable from that of a human. The Turing Test shifted the focus from

mechanical computation to the broader question of machine intelligence.

Early Machine Learning and Symbolic AI Following the Turing Test, research into machine learning and symbolic AI accelerated. In the 1950s, the perceptron model was introduced as an early neural network capable of binary classification. Around the same time, James Slagle developed an early influential AI program: a symbolic integration system capable of performing calculus operations symbolically rather than numerically. This line of work anticipated themes later formalized in decision procedures for elementary integration (Risch, 1969) and demonstrated that machines could manipulate abstract symbols to solve problems, a core idea in symbolic AI.

Summary of Key Historical Milestones

- **12th–13th Centuries:** Mechanical automata (e.g., Al-Jazari) and scholastic refinements of syllogistic logic.
- **19th Century:** Charles Babbage's Analytical Engine and Ada Lovelace's pioneering programming notes; Boole and contemporaries formalize symbolic logic.
- **Early 20th Century:** Frege, Peano, Russell, and Whitehead develop predicate calculus and logicist foundations.
- **1950:** Alan Turing's Turing Test frames the question of machine intelligence.
- **1950s:** Development of early machine learning models (perceptrons) and symbolic AI programs (e.g., Slagle's integration system).

This historical arc sets the stage for contemporary intelligent systems: programmable artifacts whose behavior is grounded in formal models, implemented on digital hardware, and increasingly trained or tuned from data. The sections that follow make the working definitions and modeling assumptions explicit.

1.2 Defining Artificial Intelligence and Intelligent Systems

Artificial Intelligence (AI) is often misunderstood as merely a collection of popular applications such as image recognition or voice detection. However, these are just subsets of a much broader field. Instead of defining AI by its famous

applications, it is more accurate to view AI as a body of collective algorithms, research, and engineering practice aimed at enabling machines to perceive their environment, perform inference, and take purposeful actions.

Core Definition of AI Following the agent-centric view of Russell and Norvig (2021), artificial intelligence studies computational agents that map percepts to actions through algorithms operating over explicit representations (state graphs, feature vectors, logical predicates, or probabilistic models) subject to domain constraints (physical limits, safety rules, resource budgets). See also Poole and Mackworth (2017) for a complementary treatment focused on agent architectures. Each model we study is evaluated on whether its assumptions support competent *perception* (information acquisition), *reasoning and decision-making* (information processing), and *action* (environment intervention), where a *percept* denotes the data received at a decision epoch (a discrete sensing-and-decision instant; e.g., sensor readings, feature vectors, linguistic tokens) and an *action* denotes the command issued to the environment or downstream system.

Many model-based systems generate hypotheses and test them, yet the field also includes purely reactive controllers (e.g., subsumption architectures in behavior-based robotics or PID loops) that optimize behavior without explicit hypothesis testing. Classic behavior-based robotics research (Brooks, 1986; Arkin, 1998) treats such controllers as intelligent agents. They satisfy the perception–action cycle even in the absence of symbolic reasoning. We flag them as boundary cases: they remain control-theoretic constructs, yet they highlight the continuum between classical control and adaptive AI systems. Throughout this book we discuss both deliberative reasoning (planning, inference, search) and reflexive intelligence (engineered feedback loops that achieve goals without symbolic reasoning), and we try to make clear which lens is being used in a given chapter.

For now, if we adopt a value-centric view of AI, we can characterize intelligent systems by the kinds of questions they help us answer. In practice, three capabilities dominate:

- Explaining the past,
- Understanding the present, and
- Predicting the future.

Framed this way, the parallel with human intelligence becomes explicit: both artificial systems and humans are judged by how well they can reconstruct what has happened, make sense of what is happening, and anticipate what is likely to happen next. For example, humans use memory and narrative to explain past events, situational awareness to understand ongoing interactions, and mental models to predict likely outcomes. Analytic systems that perform root-cause analysis in power grids or credit-risk models in finance primarily *explain the past*; monitoring systems such as anomaly detectors and online recommendation engines focus on *understanding the present*; time-series forecasters and large language models that predict the next token or utterance instantiate the *predicting the future* role. Modern AI architectures often blend these roles, but keeping the three questions in mind provides a useful lens for interpreting model behavior.

To connect this value-centric lens to concrete designs, we now make more precise what we mean by an intelligent system and how a design begins with a clearly stated problem and representation.

1.3 Intelligent Systems

An *intelligent system* is an artificial entity composed of both software and hardware components that:

- Acquire, store, and apply knowledge,
- Perceive and interpret environmental data to maintain situational awareness,
- Make decisions and act based on incomplete or imperfect information.

In contrast, an *intelligent machine* is usually a single embodied device (for example, a robot arm on a factory line) whose sensing, reasoning, and actuation are co-located. Intelligent systems can comprise multiple cooperating machines plus cloud services; intelligent machines are one concrete realization within that broader system-of-systems view.

This working definition is consistent with those used in cyber-physical systems literature and the IEEE Standards Association's descriptions of intelligent agents, emphasizing perception, cognition, and action as the three pillars of autonomy.[1]

[1] Compare with the IEEE Global Initiative on Ethics of Autonomous and Intelligent Systems,

Here, "knowledge" encompasses encoded data sets, learned model parameters, rule bases, and semantic ontologies that the system can query or update during operation. The hardware enables interaction with the environment (e.g., sensors, actuators), while the software performs reasoning and decision-making.

1.3.1 From value-centric questions to concrete designs

The three value-centric questions ("explain the past, understand the present, predict the future") only become actionable once a designer fixes a problem statement, a representation, and the constraints under which the system operates. Rather than treating these as separate case studies, we fold them into a compact design checklist that we reuse whenever it helps structure a design discussion:

1. **Problem definition.** State the task in operational terms. Example: "Detect stop signs quickly enough to enable safe braking." The definition should tell us which of the three value-centric roles dominates (here: understanding the present, plus explaining why braking events occur).

2. **Representation.** Decide how the world will be encoded numerically. Stop-sign detection uses camera images (matrices of intensities) plus metadata such as lane boundaries or GPS position; a financial recommender might rely on structured tabular data.

3. **Objectives and constraints.** Specify the metric to optimize (e.g., minimize false negatives) and the hard constraints (minimum stopping distance, latency budgets, regulatory rules). Practical implementations refine these with regions of interest, masking, or sensor fusion (LiDAR + camera) so the classifier only runs where a stop sign could plausibly appear.

These three ingredients determine what the intelligent system must sense, infer, and control. Once they are in place we can reason about the interacting components that implement the perception → reasoning → action loop.

1.3.2 Components of AI Systems: Thinking, Perception, and Action

AI systems can be decomposed into three interrelated components:

Perception: How the system senses and interprets environmental data, extracting features or state estimates.

Ethically Aligned Design, 1st ed., 2019.

Reasoning and Decision-Making: How the system combines models and control policies with learned value functions to plan actions or react in real time.

Action: How the system executes decisions to affect the environment.

Example: Autonomous Vehicle

- **Perception:** Camera captures images, which are converted into numerical arrays.
- **Thinking:** Algorithms classify objects (e.g., stop signs, pedestrians) and predict future states.
- **Action:** Vehicle control systems adjust steering, acceleration, or braking accordingly.

1.4 Case Study: AI-Enabled Camera as an Intelligent System

The design checklist above becomes concrete when we dissect a deployed system. Consider a networked camera that detects humans and escalates alarms in an industrial plant.

Checklist instantiated for the camera system

Problem + value role

Detect humans entering restricted zones (understand the present) and log footage for later audits (explain the past).

Representation

Images are streamed as $H \times W \times 3$ tensors; regions of interest and background models are maintained to suppress noise.

Objectives/constraints

Maintain < 200 ms end-to-end latency and $< 1\%$ false negatives; respect privacy/retention policies.

Hardware (perception/action)

CMOS sensor, on-board DSP/accelerator, motorized pan/tilt for re-targeting.

Software (reasoning)

YOLOv8-style detector fine-tuned on site-specific data, fused with Kalman filters for track smoothing and MPC logic that commands the pan/tilt actuator or triggers alerts.

Integration

Edge inference handles immediate reactions; metadata is sent to a cloud analytics service that enriches logs and retrains models (predicting the future by anticipating recurrent intrusion times).

This decomposition highlights the same three pillars—perception, reasoning, and action—while adding the operational nuance (latency budgets, privacy constraints) that graduate-level systems must address. Many later chapters discuss building blocks that could be used in such a pipeline: convolutional networks (Chapter 11) for visual detection, recurrent models (Chapter 12) for temporal smoothing and sequence prediction, supervised learning and calibration (Chapters 3 to 4) for reliable scoring and threshold selection, fuzzy controllers (Chapters 16 to 18) for rule-based escalation policies, and evolutionary algorithms (Chapter 19) for tuning design choices such as placement, thresholds, or hyperparameters.

An intelligent system is therefore not just the hardware or the software alone, but the *system of components* working together to perceive, reason, and act under explicit objectives.

1.5 Levels and Architectures of Intelligent Systems

Having introduced working definitions and concrete examples, we now summarize capabilities and architectural patterns that reappear across the book.

What Constitutes Intelligence in Systems? Intelligence in systems is often characterized by the ability to:

- **Perceive** and interpret inputs from the environment.
- **Reason** and make decisions based on available information.
- **Learn** from experience to improve performance.
- **Act** autonomously to achieve goals.

These capabilities can be realized in various architectures, ranging from connectionist models (e.g., neural networks) to symbolic systems and hybrid approaches.

Levels of Intelligence (as an organizing lens) Intelligence is not necessarily binary (intelligent vs. non-intelligent); rather, deployed systems combine different degrees of reactivity, deliberation, and adaptation. For the purposes of this book we use a four-layer shorthand—reactive systems (level 1), deliberative planners (level 2), adaptive learners (level 3), and meta-cognitive agents that reason about their own policies (level 4)—as an informal organizing lens rather than a strict hierarchy. It is compatible with domain-specific taxonomies (e.g., SAE Levels 0–5 for automated driving). The closing Key Takeaways return to it with representative algorithms from later chapters.

Connectionist vs. agent-based/decentralized approaches Two broad paradigms in intelligent system design are:

- **Connectionist Models:** Systems structured as interconnected processing units (e.g., neural networks) with defined input-output stages.

- **Agent-based or decentralized systems:** Collections of agents or modules that operate semi-independently, often with only local communication, such as swarm intelligence or evolutionary algorithms.

Both approaches have merits and limitations, and hybrid models often combine elements of each.

Example: Swarm Intelligence Swarm systems consist of multiple agents solving subproblems independently but collectively achieving a global objective. Each agent follows simple rules without a global world model, yet the emergent behavior can be intelligent. This contrasts with monolithic systems possessing explicit internal representations.

Swarm intelligence can be formalized via decentralized update laws of the form $x_i(t+1) = f\big(x_i(t), \{x_j(t)\}_{j\in\mathcal{N}_i}\big)$, where each agent i interacts only with its neighborhood $\mathcal{N}_i$. Similar update patterns appear again in Chapter 19, but we do not focus on stability proofs in this book.

Examples of Input and Output Variables in Dynamic Systems To ground these ideas, consider input/output sketches from systems readers often encounter in labs or industry. Each pairs raw sensory cues with actuator or decision outputs.

- **Autonomous quadrotor:**
  - *Inputs:* Inertial measurement unit (IMU) rates, barometer/altimeter, camera or LiDAR features, GPS fixes.
  - *Outputs:* Motor thrust commands and attitude setpoints that regulate yaw/pitch/roll and track waypoints.
- **Smart microgrid:**
  - *Inputs:* Load forecasts, solar/wind availability, electricity prices, state-of-charge estimates for batteries.
  - *Outputs:* Dispatch setpoints (generator outputs, battery charge/discharge, demand-response signals) that balance stability, cost, and emissions.
- **Building HVAC controller:**

  - *Inputs:* Zone temperature/CO_2/humidity sensors, occupancy estimates, outdoor weather feeds.
  - *Outputs:* Fan speeds, damper positions, valve openings, heat-pump setpoints—tunable levers to meet comfort and energy targets.

- **Robot-assisted surgery:**
  - *Inputs:* Endoscopic vision, force/torque sensing at instruments, surgeon console commands.
  - *Outputs:* Precise tool trajectories, force limits, and safety interlocks that respect tissue constraints.

These vignettes echo a common pattern: intelligent systems fuse heterogeneous sensors to produce calibrated control or decision signals under safety, comfort, or performance constraints.

Emotions as Utility Signals

From a design perspective, emotions can be viewed abstractly as changes in an agent's internal utility or value function: positive affect corresponds to utility gains, negative affect to losses, and social emotions to comparisons between agents' utilities. Artificial systems can mimic this by modulating learning rates, exploration pressure, or safety margins in response to internal "frustration" or "satisfaction" signals without presupposing rich phenomenology. This book treats this view strictly as a modeling device for embedding motivational signals into controllers; affective computing and cognitive science work with much richer state representations than we use here.

Key Characteristics of Intelligent Systems Building on the examples above, we summarize the essential capabilities that characterize an intelligent system:

1. **Sensory Perception:** The system must be able to receive and interpret inputs from its environment, which may be in various forms such as numerical data, images, sounds, or tactile signals.
2. **Pattern Recognition and Learning:** The system should identify pat-

terns within the input data, including hidden or subtle features, and improve its performance over time by learning from experience.

3. **Knowledge Retention:** Acquired knowledge must be stored and utilized for future decision-making.
4. **Inference from Incomplete Information:** The system should be capable of drawing conclusions and making decisions even when presented with partial or approximate data.
5. **Adaptability:** It must handle unfamiliar or novel situations by generalizing from prior knowledge and adapting its behavior accordingly.
6. **Inductive Reasoning:** The system should be able to generalize patterns from observed examples—i.e., infer general rules or hypotheses from specific data instances (e.g., learn a classifier from labeled data). This differs from applying pre-written conditional logic; induction discovers the rules, whereas conditional statements merely execute them.

Intelligent Systems as Decision Makers At the core, intelligent systems perform a mapping from inputs to outputs, where the outputs represent decisions or actions influenced by the system's internal understanding or model of the environment. Formally, if we denote the input vector by $\mathbf{x} \in \mathcal{X}$ and the output vector by $\mathbf{y} \in \mathcal{Y}$, then an intelligent system implements a function

$$\mathbf{y} = f(\mathbf{x}; \theta), \tag{1.2}$$

where θ represents internal parameters or knowledge that may evolve over time through learning. This abstraction is shared by the diverse examples seen so far: they all ingest data, transform it through a parameterized mapping, and emit decisions or control signals. In this book we primarily use the system-level language (mapping under constraints), and we use "machine" when embodiment and actuation are the point. Because the chapter alternates between these viewpoints, we briefly clarify terminology and the limits of the language we use.

1.6 Intelligent Systems and Intelligent Machines

Terminology Clarification

- **Intelligent System:** A computational system (encompassing its hardware, software, and data interfaces) that perceives its environment, processes information, and acts autonomously or semi-autonomously (with limited human oversight or shared control).
- **Intelligent Machine:** A physical instantiation of an intelligent system, often embodied as a robot or automated device.

The terms are related but not identical; intelligent machines are a subset of intelligent systems, typically emphasizing the physical embodiment.

Behavior, Not Components The word *intelligent* is inevitably anthropocentric: in practice, we judge intelligence through observed behavior and performance under constraints. Motors, sensors, and circuits are enabling components, but they are not "intelligent" on their own. Intelligence emerges from how the full system processes inputs, maintains state, and selects actions.

Intelligence is also not synonymous with optimality. Many deployed systems are approximate, noisy, or biased, yet they can still be meaningfully analyzed and improved as goal-directed agents. In this book, "intelligent" is therefore used in an engineering sense: a system that maps percepts to actions with a design intent, and that can be evaluated against explicit objectives and constraints.

Examples Robots developed by Boston Dynamics (e.g., quadrupeds) illustrate how feedback control, state estimation, and trajectory planning can produce behaviors that humans interpret as intelligent (balance recovery, robust locomotion, disturbance rejection) even though the system has no intrinsic understanding or feelings. Voice-activated assistants and robots provide a second common example: they appear intelligent because they can condition actions on language inputs, maintain limited context, and complete tasks that align with user intent.

Consciousness and Intelligence While machines can exhibit intelligent behaviors, the question of whether they possess consciousness or self-awareness remains open and is a subject of ongoing research and philosophical debate.

In this book we treat consciousness operationally:
we focus on meta-cognition (self-monitoring of one's own decision process) rather

than phenomenal awareness. This keeps the discussion tied to observable, designable behaviors rather than philosophical claims.

Author's note: "subject of its own thought"

Strong machine intelligence, in my view, means a system can make itself the subject of its own thought. It keeps self-models (confidence monitors, policy auditors, explanation traces) to critique and revise its reasoning, not just its predictions.

The rest of this book uses the simpler four-layer taxonomy (reactive → deliberative → adaptive → meta-cognitive), but the "subject of its own thought" view motivates why Level 4 systems need special care in design and governance.

1.7 Levels, Meta-cognition, and Safety

This book uses levels of intelligence as an organizing lens rather than a formal taxonomy. The four levels introduced above (reactive, deliberative, adaptive, and meta-cognitive) are meant to clarify what a system can do, what it must represent, and what kinds of failures are plausible. For a working definition of AI and intelligent systems, see Section 1.3.

Meta-cognition (Operational View) In this book, meta-cognition refers to a controller's ability to monitor, assess, and revise its own reasoning policies. In practice this can look like confidence monitors, audits of decision traces, and bounded self-correction loops rather than unconstrained self-modification.

Implications and Risks If a system can improve its own utility autonomously and rapidly, it may induce competitive dynamics in which improving one utility degrades another's. This occurs in multi-agent settings (competing organizations or robots) and in multi-objective optimization when safety objectives conflict with performance. These scenarios motivate conservative design and governance, especially as systems move from adaptive learning to self-monitoring and policy revision.

Designing Safe Intelligent Systems One practical mitigation is to build systems that:

- Keep auditable records of their decisions and updates.
- Perform self-inspection and error analysis.
- Backtrack and self-correct within explicit, designer-defined bounds.

Such systems can improve without uncontrolled self-modification: policy updates are gated by testable criteria, and any self-editing of code or reward functions proceeds only through approved interfaces.

Reader's guide. The remainder of this chapter is practical: who the book is for, how chapters fit together, and how to navigate recurring structure. Notation and reading conventions are collected in the front matter (see *Notation and Conventions*).

1.8 Audience, Prerequisites, and Scope

This material has been rewritten to stand on its own as a book. It surveys the design and analysis of intelligent systems along two main strands:

- data-driven models for prediction and decision making (linear models, kernels, deep networks, sequence models, Transformers);
- soft-computing and search methods (self-organizing maps, fuzzy systems, evolutionary and genetic algorithms).

The emphasis is on breadth with enough mathematical depth that you can relate ideas across chapters rather than treating each technique in isolation.

The book also maintains a deliberate dual emphasis: representation learning with neural and kernelized models on one hand, and soft-computing approaches (fuzzy systems and evolutionary optimization) on the other. This balance keeps robustness, interpretability, and optimization themes all in view rather than treating deep networks in isolation.

The assumed background is undergraduate calculus and linear algebra (vectors, matrices, eigenvalues) and basic probability and statistics. No prior dedicated AI or machine-learning course is assumed: key ideas such as losses, optimization, gradient descent, backpropagation, kernels, fuzzy operators, and evolutionary operators are introduced from first principles when they first appear. Familiarity with signals and systems, and with linear time-invariant (LTI) models in particular, is helpful for the sequence-modeling and control-oriented parts

of the book; Appendix A (*Linear Systems Primer*) provides a concise refresher.

1.9 Roadmap and Reading Paths

Figure 1 summarizes the narrative arc of the book: a core supervised path (linear and logistic regression to MLPs to CNNs to RNNs), a branch through competitive learning and fuzzy inference for rule-based reasoning, and a parallel thread on optimization culminating in evolutionary computing. Early on, Chapter 2 provides a complementary symbolic-search perspective so we can contrast "intelligence via transformations" with ERM-based modeling. Chapters cross-reference one another so you can skim the path most relevant to your project and return for foundational refreshers as needed.

Readers arrive with different goals. The roadmap is intentionally a dependency graph rather than a single linear track; the following paths are common starting points:

1. **ML-focused path:** Chapters 3 to 4 → Chapters 6 to 7 → Chapters 11 to 13 → Chapter 14.
2. **Control/systems path:** Chapters 1 to 3 → Chapter 10 → Chapters 15 to 18 → Chapter 19.
3. **Soft-computing path:** Chapters 1 to 3 → Chapters 15 to 18 with optional detours to Chapter 6 and Chapter 19.

1.10 Using and Navigating This Book

- **Before each chapter:** skim the Learning Outcomes and check where it sits on the Roadmap.

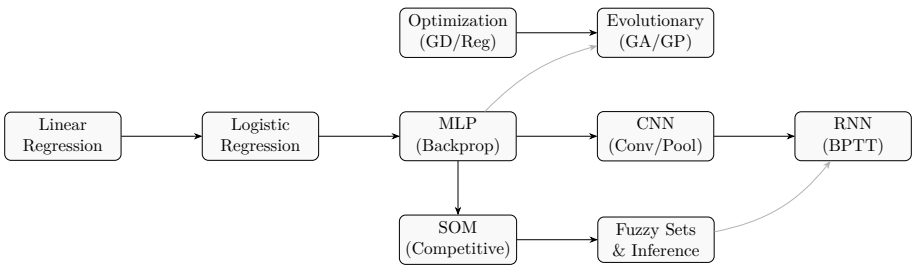

Figure 1: Schematic: Roadmap (core supervised path; SOM/fuzzy; optimization/evolutionary).

- **While reading:** follow cross-referenced figures/equations and pause at the short checkpoints and worked examples.
- **After each chapter:** review the Summary and Common Pitfalls; revisit the pseudocode and attempt the Exercises.
- **Keep the front matter close:** the Notation and Conventions section defines symbols reused throughout the book; cross-references point back to it when conventions matter.

Conventions and reading aids. The front matter summarizes notation and common conventions, and it also includes a short guide to reading figures and recurring box styles.

Key takeaways

- Intelligent systems integrate perception, decision, and action; we study both model-based and control-based realizations.
- The system–machine distinction is mostly about embodiment: intelligent machines are physical instances of intelligent systems.
- "Levels" are an organizing lens, and meta-cognition is treated operationally (self-monitoring and bounded self-correction), not philosophically.

Exercises and lab ideas

- Pick one engineered system you know (e.g., a recommender, a robot, a control loop) and identify its *percepts*, *internal representation*, and *actions*.
- For one section of this chapter, rewrite the core definition in your own words and list one concrete failure mode the definition helps you anticipate.
- Use Figure 1 to choose a reading path and write down which two chapters you will skim first (and why).

Where we head next. To make the abstract vocabulary of "intelligence" concrete, we start with a deliberately small case study: symbolic integration

as transformation search (Chapter 2). It is a crisp example of a system that decomposes a problem, applies safe rules, branches heuristically, backtracks, and certifies progress or failure with residual checks—capabilities that reappear later, often in data-driven form.

References. Full citations for works mentioned in this chapter appear in the book-wide bibliography.

2 Symbolic Integration and Problem-Solving Strategies

Learning Outcomes

- Decompose symbolic integration problems into safe vs. heuristic transformations and understand when each is appropriate.
- Trace the transformation-tree search (state save/restore, heuristics, termination) and connect it to broader notions of intelligent problem solving.
- Contrast symbolic transformation search with data-driven modeling pipelines and identify what each paradigm contributes.

Chapter 1 treated "intelligence" operationally: a system should represent a problem, choose actions, and verify or correct itself under constraints. This chapter makes that concrete with a deliberately small example—symbolic integration—where the system's actions are algebraic transformations. The roadmap in Figure 1 situates this symbolic strand alongside the data-driven path.

Design motif

Meaning-preserving moves plus a mechanical check: if $F(x)$ is proposed as an antiderivative of $f(x)$, then differentiating should recover the integrand on the declared domain, i.e., $F'(x) = f(x)$ (equivalently, $F'(x) - f(x) = 0$).

We study symbolic problem solving through the lens of *integration-by-transformation*: preserve meaning while rewriting an expression into a form that exposes a solution. The point is not the catalog of tricks in isolation, but

the system-level pattern: explicit representations, meaning-preserving actions, heuristic branching with backtracking, and a clear goal test that certifies correctness.

It helps to separate transformations into two tiers: reliable moves that never change the antiderivative class (factoring constants, linear substitutions, polynomial division) and heuristic moves that may succeed only for certain structures. A practical policy is to apply every reliable step first, then branch judiciously through the heuristic catalog while keeping enough state to backtrack.

Aside: the Risch algorithm

The Risch algorithm (Risch, 1969) decides whether an elementary antiderivative exists for a large class of integrands by reducing the problem to algebra over differential fields. Unfortunately its implementation is intricate, requires case explosions, and still leaves many useful nonelementary functions unresolved. Practical computer algebra systems therefore augment Risch-style decision procedures with heuristic transformations—the focus of this chapter—to keep runtimes bounded and to return human-readable answers when possible.

2.1 Context and Motivation

Consider the task of solving an integral of the form

$$\int f(x)\,dx,$$

where $f(x)$ may be a complicated function. Traditional approaches often rely on consulting integral tables or applying well-known formulas. For example, integrals such as

$$\int \frac{1}{x}\,dx = \ln|x| + C.$$

These are straightforward and can be solved by direct lookup or simple substitution.

However, many integrals encountered in practice do not match any entry in standard integral tables, nor do they succumb easily to elementary techniques. The integration-by-transformation view treats this as a search problem: propose meaning-preserving rewrites, backtrack when a branch gets harder, and verify

candidates mechanically by differentiation.

2.2 Problem Decomposition and Transformation

A key insight in tackling complex integrals is to *reduce* the problem into manageable subproblems. This involves applying *transformations* to rewrite the integral into a form that is either directly solvable or closer to known forms.

Safe Transformations We define *safe transformations* as invertible substitutions that allow back-substitution: if $u = \phi(x)$ and F_u is an antiderivative of $T[g](u)$, then $F_u \circ \phi$ differentiates back to $g(x)$. Safe transformations are algebraic substitutions or factorings that survive reversal. Examples include:

- **Constant factor extraction:** If G is an antiderivative of g, then ag has antiderivative aG; differentiating confirms $\frac{d}{dx}[aG(x)] = ag(x)$.
- **Linear substitution:** Let $u = ax + b$ with $a \neq 0$. Differentiating gives $du = a\,dx$ and hence $dx = \frac{du}{a}$; substituting shows that

$$\int f(ax+b)\,dx = \frac{1}{a}\int f(u)\,du.$$

  This is the standard change-of-variables formula.

- **Polynomial division:** If $p(x)$ and $q(x)$ are polynomials with $\deg p \geq \deg q$, then perform polynomial division:

$$\frac{p(x)}{q(x)} = s(x) + \frac{r(x)}{q(x)},$$

  where $\deg r < \deg q$. Linearity of integration lets us integrate $s(x)$ term-by-term, while the proper fraction $\frac{r(x)}{q(x)}$ can be addressed via partial fractions or further substitutions, yielding an equivalent antiderivative.

These transformations are *safe* because they always preserve the integral's value and simplify the problem without introducing ambiguity. When a substitution transforms an integral over $x \in [0,1]$ into $\int_0^1 u^b(1-u)^c\,du$ with $b, c > -1$, the resulting definite integral evaluates to a Beta function $B(b{+}1, c{+}1)$; the Beta identity applies to that definite integral on $[0,1]$, and it is therefore customary to fall back on it when an elementary antiderivative is unavailable.

Example: Applying Safe Transformations Suppose we have an integral of the form

$$\int a \cdot x^b (1-x)^c \, dx,$$

where a, b, c are constants. A safe transformation might be to factor out the constant a and then consider substitutions or binomial expansions that reduce the powers to known integrals (e.g., Beta-function evaluations when b and c are integers).

2.3 Limitations of Safe Transformations

After exhaustively applying all safe transformations, we may still encounter integrals that do not match any known solvable form. At this point, the system must decide whether the problem is solvable by known methods or if alternative strategies are necessary.

2.4 Heuristic Transformations

When safe transformations fail to yield a solution, we turn to *heuristic transformations*, which are not guaranteed to succeed but often provide a path forward. These heuristics are based on experience, pattern recognition, and mathematical intuition.

Definition Heuristic transformations are problem-solving *tricks* or *strategies* that attempt to rewrite the integral into a solvable form by exploiting structural properties of the integrand. They may involve:

- Trigonometric identities and substitutions, e.g., using relationships among $\sin x$, $\cos x$, $\tan x$, $\cot x$, $\sec x$, and $\csc x$.
- Algebraic manipulations that simplify complicated expressions.
- Variable substitutions that transform the integral into a standard form.
- Recognizing patterns such as functions of $10x$ or other scaled arguments and applying appropriate scaling substitutions (e.g., if the integrand contains $f(cx)$, introduce $u = cx$ so that the scale factor is absorbed).

Example: Trigonometric Heuristics Consider an integral involving sine and cosine:

$$\int \frac{\sin x}{\cos x}\, dx.$$

Recognizing that $\sin x / \cos x = \tan x$, we can rewrite the integral as

$$\int \tan x\, dx = -\ln|\cos x| + C.$$

which is a standard integral with the constant of integration explicitly noted.

Similarly, if the integrand involves expressions like $\sin^2 x + \cos^2 x$, we can use the Pythagorean identity to simplify.

Heuristics as a Form of Intelligence The use of heuristic transformations reflects a form of mathematical intelligence: the ability to recognize patterns, apply non-obvious substitutions, and creatively manipulate expressions to reach a solution. Unlike safe transformations, heuristics may fail or lead to dead ends, but they expand the problem-solving repertoire beyond mechanical procedures.

Absolute values and branches

- **Square roots:** specify the sign/branch. If you drop $|\cdot|$, restrict the substitution interval so the sign is fixed (e.g., $\cos y \geq 0$ on $y \in (-\pi/2, \pi/2)$ so $\sqrt{1-\sin^2 y} = \cos y$).
- **Logarithms:** default to $\log|f(x)|$ unless you guarantee f keeps one sign on the declared domain.
- **Arctrig/hyperbolic inverses:** state principal values and any periodicity you rely on for back-substitution.

2.5 Summary of the Approach

The overall strategy for symbolic integration can be summarized as follows:

1. **Apply all safe transformations** to simplify the integral and attempt to match known solvable forms.
2. **Re-evaluate the transformed integrand** to identify structural cues (symmetry, polynomial degree, trigonometric patterns).

3. **Choose among multiple transformation paths** by comparing simple cost heuristics such as expression-tree depth, number of nonzero coefficients, or anticipated integration rules.

4. **Fallback to heuristics and backtracking** when safe transformations stall, maintaining a stack of previous states to enable systematic exploration.

Cost heuristic. Score candidates by a triple: tree depth, number of nonlinear operators, and symbol count. The nonlinearity term counts transcendental nodes (e.g., trigonometric, exponential, logarithmic), while the symbol count tallies AST nodes excluding simple literals such as $-1, 0, 1$. Prefer branches that reduce or preserve this triple, and break ties toward rules with known templates (e.g., partial fractions, reduction formulas).

2.6 Heuristic Transformations: Revisiting the Integral with $1 - x^2$

Recall the integral under consideration:

$$\int \frac{4}{(1-x^2)^{5/2}}\,dx. \tag{2.1}$$

For real-valued integration we restrict attention to $|x| < 1$, ensuring the denominator $(1-x^2)^{5/2}$ is well-defined and nonzero on the interval of interest.

When encountering expressions involving $1 - x^2$, a classical heuristic substitution is:

$$x = \sin y,$$

which leverages the Pythagorean identity:

$$1 - \sin^2 y = \cos^2 y.$$

Applying this substitution transforms the integral into a trigonometric form that is often easier to handle.

Step 1: Substitution and Differential Set

$$x = \sin y \quad \Longrightarrow \quad dx = \cos y \, dy.$$

We take $y = \arcsin x$ with $y \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ so that the substitution remains bijective on the domain $|x| \leq 1$, and note $\cos y \geq 0$ on this interval so that $\sqrt{1 - \sin^2 y} = \cos y$ is consistent with the chosen branch.

Substituting into (2.1) and using $dx = \cos y \, dy$ yields

$$\begin{aligned} \int \frac{4}{(1-x^2)^{5/2}} \, dx &= \int \frac{4}{(1-\sin^2 y)^{5/2}} \cos y \, dy \\ &= \int \frac{4 \cos y}{(\cos^2 y)^{5/2}} \, dy \\ &= 4 \int \cos^{-4} y \, dy \\ &= 4 \int \sec^4 y \, dy. \end{aligned}$$

The intermediate step $\cos y \cdot \cos^{-5} y = \cos^{-4} y$ is made explicit so the exponent arithmetic is transparent.

Thus, the integral reduces to

$$4 \int \sec^4 y \, dy. \tag{2.2}$$

Step 2: Choosing the Next Transformation At this stage, two common safe transformations are available:

- Express $\sec^4 y$ in terms of $\tan y$, using the identity $\sec^2 y = 1 + \tan^2 y$, and then perform substitution $u = \tan y$ with $du = \sec^2 y \, dy$.
- Use reduction formulas for powers of secant directly, e.g.,

$$\int \sec^n y \, dy = \frac{\sec^{n-2} y \tan y}{n-1} + \frac{n-2}{n-1} \int \sec^{n-2} y \, dy, \quad n > 1.$$

Standard reduction formulas provide a deterministic alternative if the substitution path is judged too costly.

The choice between these paths is nontrivial, especially for an automated

system. Humans often pick the substitution $u = \tan y$ intuitively because it simplifies the integral, but a machine requires a deterministic decision rule.

Step 3: Functional Composition and Path Selection To automate the choice, the system evaluates the *functional composition* of the integral expressions along each path (e.g., measuring expression-tree depth, symbolic coefficient growth, or the number of distinct functions involved):

- **Path 1:** Substitution $u = \tan y$ reduces the integral to a polynomial in u, which is straightforward to integrate.
- **Path 2:** Direct reduction of $\sec^4 y$ may involve more complex recursive steps.

From a cost perspective, Path 1 is cheaper and more direct, so the system prioritizes it. However, if this path fails to yield a solution, the system must backtrack and attempt Path 2.

Two safe options from here

- **(a) Substitution $u = \tan y$.** Since $\sec^4 y\,dy = \sec^2 y\,(\sec^2 y\,dy)$ and $\sec^2 y = 1 + \tan^2 y$, set $u = \tan y$, $du = \sec^2 y\,dy$:

$$\begin{aligned} 4\int \sec^4 y\,dy &= 4\int (1+u^2)\,du \\ &= 4\left(u + \tfrac{u^3}{3}\right) + C \\ &= 4\tan y + \tfrac{4}{3}\tan^3 y + C. \end{aligned}$$

- **(b) Reduction formula.** For even $n > 1$,

$$\int \sec^n y\,dy = \frac{\sec^{n-2} y \tan y}{n-1} + \frac{n-2}{n-1}\int \sec^{n-2} y\,dy.$$

  Applying this with $n = 4$ gives $\int \sec^4 y\,dy = \tan y + \frac{1}{3}\tan^3 y + C$, reproducing the same primitive without the u-substitution.

Back-substitution and check Using $\tan y = \dfrac{x}{\sqrt{1-x^2}}$, both paths yield:

$$F(x) = 4\left[x(1-x^2)^{-1/2} + \frac{x^3}{3(1-x^2)^{3/2}}\right] + C.$$

Differentiating term by term shows $F'(x) = 4(1-x^2)^{-5/2}$ on $|x| < 1$. Outside $(-1, 1)$ the principal-branch integrand is complex-valued; a real continuation rewrites the integral as $\int 4(x^2-1)^{-5/2}dx$ with $x = \cosh t$.

Pattern rule For integrals of the form $\int (1-x^2)^{-k-1/2}dx$, the substitution $x = \sin y$ reduces them to $\int \sec^{2k} y\, dy$; apply the even-power reduction accordingly. This is why the $1 - x^2$ pattern triggers the trigonometric branch.

2.7 Example: Solving an Integral via Transformation Trees

The worked integral above shows how the search explored multiple branches (e.g., $x = \sin y$ vs. $x = \tanh u$) while respecting the declared domain. Heuristic branches that do not fit the domain are pruned; competing safe branches (substitution vs. reduction) converge to the same antiderivative. The key insight is that solving integrals can be viewed as traversing a *decision tree* of transformations, with goal tests supplied by residual checks and domain conditions.

2.8 Transformation Trees and Search Strategies

Definition: A **transformation tree** is a conceptual structure representing all possible sequences of transformations applied to an expression in an attempt to solve or simplify it.

- Each node corresponds to a state of the expression.
- Edges correspond to transformations (safe or heuristic).
- Leaves correspond to either solved expressions or dead ends (no solution).

Figure 2 shows the actual tree explored for $\int \frac{4}{(1-x^2)^{5/2}}\, dx$. Solid branches denote safe algebraic steps (guaranteed progress), while dashed branches illustrate heuristic substitutions that may fail and trigger backtracking. Computer algebra systems follow similar playbooks (Bronstein, 2005; Risch, 1969): a Risch-style decision core handles provably solvable cases, while a curated bank of heuris-

tics (pattern rewrites, rational substitutions, special-function fallbacks) explores auxiliary branches with explicit depth/time budgets.

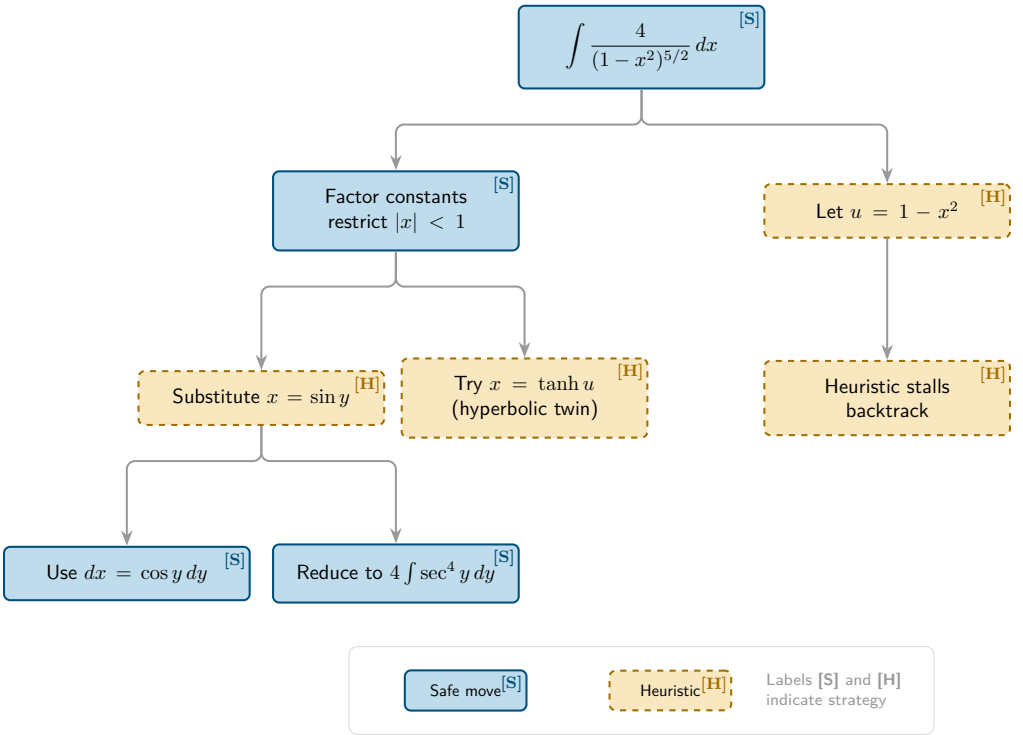

Figure 2: Schematic: Transformation tree for the running example integral; badges **[S]**/**[H]** mark safe vs. heuristic moves; the dashed branch mirrors the sine substitution.

Example: For the integral problem, the root node is the original integral. From there, we branch into applying different substitutions or algebraic manipulations, such as

$$\begin{aligned}\text{Apply substitution } u = \tan(x) &\Rightarrow \text{integration by parts} \\ &\Rightarrow \text{inverse trig identities} \Rightarrow \cdots\end{aligned}$$

Safe vs. Heuristic Transformations:

- **Safe transformations** are guaranteed to preserve equivalence and progress towards a solution.

- **Heuristic transformations** may or may not lead to a solution; they are attempts that carry risk but can be beneficial.

Backtracking: If a branch leads to no solution, the system must backtrack to a previous node and try alternative transformations. This requires the ability to:

- *Freeze* the current state before branching.
- *Restore* previous states upon failure (e.g., by pushing serialized expression trees and associated metadata onto a stack for later reinstatement).

In practice this corresponds to pushing serialized expression trees (i.e., deep copies of the tree structure together with any transformation metadata) onto a stack so they can be reinstated after unsuccessful exploratory steps.

2.9 Algorithmic Outline for Symbolic Problem Solving

The general algorithm for solving symbolic problems such as integrals can be summarized as follows:

1. **Define the goal:** For example, express the integral in terms of known functions from a table.
2. **Enumerate transformations:** List all possible safe and heuristic transformations applicable to the current expression.
3. **Apply safe transformations:** Attempt all safe transformations and check if the problem is solved.
4. **If not solved, apply heuristic transformations:** Attempt heuristic transformations to explore alternative paths; common template hits include $\int f'(x)/f(x)\,dx \to \log|f(x)| + C$ and $\int r'(x)e^{r(x)}\,dx \to e^{r(x)} + C$.
5. **Branch and backtrack:** For each transformation, branch the search tree. If a branch fails, backtrack and try other branches.
6. **Use heuristics to guide search:** For example, use functional composition depth or cost metrics to prioritize branches.
7. **Terminate cleanly:** Stop when a closed-form antiderivative is found, or when depth/time budgets are exceeded without success; optional numeric

residual tests can accept approximate solutions.

Note: This approach resembles a *greedy search* with backtracking, but it does not guarantee an optimal or even successful solution in all cases.

Transformation-tree search (pseudocode)

```
function SolveIntegral(f0,
                       domain=(-1,1),
                       depth_limit=8,
                       time_limit=2s,
                       eps_abs=1e-6,
                       eps_rel=1e-6,
                       samples=24):
    stack <- [(f0, domain, empty_history, depth=0)]
    start <- clock()
    best  <- {status="fail",
              F=None,
              residual=None,
              history=[],
              domain=domain}
    while stack not empty:
        current, dom, history, depth <- pop(stack)
        if clock() - start > time_limit
           or depth > depth_limit:
            continue
        if passes_residual_test(current, f0, dom,
                                eps_abs, eps_rel, samples):
            res = residual(current, f0, dom, samples)
            return {status="closed_form",
                    F=current,
                    residual=res,
                    history=history,
                    domain=dom}
        safe, heuristic <- enumerate_transforms(current, dom)
        for T in safe:
            g, new_dom <- apply(T, current, dom)
            push(stack,
                 (g, new_dom, history + [T], depth+1))
        for H in heuristic (ordered by cost)
             when depth+1 <= depth_limit:
            g, new_dom <- apply(H, current, dom)
            push(stack,
```

```
                (g, new_dom, history + [H], depth+1))
    return best
```

The pseudocode mirrors the narrative: record the original integrand f_0, track the current domain/branch, apply safe transforms eagerly, explore heuristic branches within time/depth budgets, and only accept a candidate antiderivative once a sampled residual check (absolute or relative) passes on points inside the declared domain; return a clear status/history/domain either way.

Residual test implementation. At each candidate F, sample K points inside the current domain but away from singularities and branch points, and form $R = \max_i |F'(x_i) - f_0(x_i)|$. Accept if $R \le \varepsilon_{\text{abs}}$ or $R/(1 + \max_i |f_0(x_i)|) \le \varepsilon_{\text{rel}}$. Automatic differentiation or a high-order finite difference (step $h = O(\sqrt{\varepsilon_{\text{machine}}})$) keeps the check numerically stable. As substitutions shrink the domain, shrink the sample set and log the updated interval alongside the history.

Termination policies and numeric fallbacks

- **Budgeting:** Cap depth, number of heuristic branches, and runtime (e.g., depth limit $D = 8$, two-second wall clock). When limits are reached, report "no elementary antiderivative within budget D" rather than looping forever.
- **Residual checks:** Differentiate candidate antiderivatives symbolically and numerically. Sample points inside the declared domain (away from poles) and accept only if $\max_i |F'(x_i) - f(x_i)| \le \varepsilon_{\text{abs}}$ or the relative tolerance passes; otherwise prune or refine before returning.
- **Numeric escape hatch:** Switch to adaptive quadrature (e.g., Gauss–Kronrod) once symbolic attempts fail; return both the numeric estimate and the failed transformation history so users can adjust heuristics, noting that the numeric value is not a closed form.
- **Domain reminders:** When substitutions shrink domains (e.g., $x = \sin y$ enforces $|x| \le 1$), log the restriction and branch choice so the numeric fallback samples within the valid range and the report is reproducible.

Worked example: Beta template vs. numeric fallback Consider the Beta integral

$$I(a, b) = \int_0^1 x^{a-1}(1 - x)^{b-1}\,\mathrm{d}x, \qquad a, b > 0.$$

Safe transformations (factor constants, recognize the Beta template) immediately identify the elementary value $B(a, b) = \Gamma(a)\Gamma(b)/\Gamma(a + b)$. By contrast, the perturbed integral

$$\int_0^1 x^{a-1}(1 - x)^{b-1} \log(1 + x)\,\mathrm{d}x$$

fails the template check after all safe moves. The solver therefore (i) records the unmet template, (ii) pushes a heuristic branch such as differentiation under

Table 1: Table: Transformation toolkit (safe vs. heuristic). Preconditions keep domains/branches explicit (e.g., restrictions like "x in (-1,1)" for square-root expressions); principal branches unless noted.

	Safe	**Heuristic**
Constant factor	$\int a\,g(x)\,dx = a\int g(x)\,dx$ (no domain change).	Completing the square before attempting trig substitutions; track any resulting branch cuts.
Linear substitution	$u = ax + b,\ dx = du/a$ with $a \neq 0$, invertible on the stated interval.	Trigonometric substitutions $x = \sin u,\ x = \tan u,\ t = \tan(x/2)$ with domains $u \in (-\frac{\pi}{2}, \frac{\pi}{2})$ or stated principal branches.
Polynomial division / partial fractions	Split improper rational functions into polynomial + proper fraction where denominators stay nonzero on the domain.	Rationalising substitutions such as $x = 1/u$ or $x = u^2$ to expose hidden symmetry; avoid zeros/poles introduced by the map.
Log-derivative pattern	$\int f'(x)/f(x)\,dx = \log\lvert f(x)\rvert + C$ when $f(x) \neq 0$ on the domain.	Template lookups (Beta/Gamma forms with parameter sign conditions, exponential-times-polynomial motifs, etc.).

the integral sign, and (iii) if the branch exceeds time/depth budgets, falls back to adaptive quadrature with the reported residual $|I_{\text{numeric}} - I_{\text{candidate}}|$. This concrete pattern—try Beta/Gamma reduction, else return a certified numeric answer—embodies the policy described in the termination box.

Failure path with certified numeric residual. Setting $a = \frac{3}{2}, b = 2$ in the perturbed integral above illustrates the full fallback. Safe moves reduce the plain Beta integral to $B(3/2, 2) = 4/15$, but the extra $\log(1 + x)$ term triggers every heuristic branch (integration by parts, differentiation under the integral sign, series expansion) without yielding a closed form before the default depth limit $D = 8$. The solver then hands the integrand to an adaptive Gauss–Kronrod routine, which returns $I_{\text{numeric}} \approx 0.0915453885$ with an internal error certificate $< 3 \times 10^{-7}$; this is a certified quadrature value rather than a closed form. The

residual check

$$|I_{\text{numeric}} - I_{\text{previous refine}}| \leq 3 \times 10^{-7}$$

is attached to the report along with the failed transformation history, making it explicit that no elementary antiderivative was located within the allotted budget even though a numerically reliable answer exists.

2.10 Discussion: What this example illustrates

Under the operational framing in Chapter 1, the integrator exhibits several ingredients associated with intelligent problem solving: it maintains an explicit state (the current expression), chooses actions (transformations), manages contingencies (branching and backtracking), and verifies results with a crisp goal test (differentiate and check the residual). At the same time, it is limited: it does not learn new transformations from data, and its effectiveness depends on a human-designed library of moves and heuristics.

Not every heuristic is helpful. For instance, applying $x = \tan y$ to $\int (1 + x^2)^{3/2}\,dx$ looks attractive because $1 + \tan^2 y = \sec^2 y$, yet it transforms the problem into $\int \sec^5 y\,dy$, which is more complicated than the original integral. In a transformation-tree implementation this branch simply backtracks and explores alternatives (e.g., $x = \tanh u$ for $|x| < 1$ or $x = \cosh u$ for $|x| > 1$), underscoring why explicit search discipline and residual checks are essential.

This contrast helps position the data-driven chapters that follow, where the system's "actions" are parameter updates guided by loss functions and validation checks.

Connection to statistical learning. Symbolic integration is a clean playground for thinking about representations, action sequences, and verification. In data-driven modeling, the objects change (datasets, models, and losses), but the system-level pattern is similar: choose a hypothesis class, optimize an objective under resource constraints, and validate that the result generalizes.

Connection: transformation search vs. empirical risk minimization

- **Goal test:** residual check $\max_{x \in S} |F' - f| \leq \varepsilon$ vs. performance on held-out data.
- **Inductive bias:** safe/heuristic precedence vs. model class and regularization that shape what is learnable.
- **Budget:** depth/time limits vs. compute/epoch budgets and early stopping.

Key takeaways

- Symbolic integration is a compact example of a goal-driven system: represent state, apply meaning-preserving actions, and verify outcomes.
- Safe moves encode guaranteed transformations; heuristic moves trade certainty for coverage and require backtracking discipline.
- Residual checks act as a crisp goal test: differentiate a candidate and measure whether it agrees with the original integrand on the declared domain.

Exercises and lab ideas

- Implement a minimal example from this chapter and visualize intermediate quantities (plots or diagnostics) to match the pseudocode.
- Stress-test a key hyperparameter or design choice discussed here and report the effect on validation performance or stability.
- Re-derive one core equation or update rule by hand and check it numerically against your implementation.

Where we head next. For the data-driven thread (datasets, objectives, diagnostics, and classification), proceed to Chapters 3 and 4. For nonlinear function classes and nonconvex training dynamics, proceed to Chapters 5 to 6 (and onward).

References. Full citations for works mentioned in this chapter appear in the book-wide bibliography.

3 Supervised Learning Foundations

Learning Outcomes

- Formalize datasets, hypotheses, and empirical risk minimization (ERM) with consistent notation used in Chapters 3 to 4.
- Compare common regression/classification losses and regularizers, understanding when to prefer each.
- Diagnose under/overfitting with data splits, learning curves, and bias–variance reasoning; use these diagnostics to guide model selection and regularization.

Chapter 2 illustrated a non-statistical lens: solve problems by transformation search, with explicit goal tests. We now switch to the data-driven lens. The roadmap in Figure 1 marks this as the core supervised strand.

Building intelligent models is an imprecise science. If we know the relationship between the input and the output, there is no need to infer it: Celsius and Fahrenheit are linked by a simple formula, and many physical laws provide direct mappings from one quantity to another. In the problems that motivate machine learning, the mapping is unknown, messy, or only partially understood, so we settle for an approximation. That approximation might be statistical (learned from data), rule-based (encoded from experience), biologically inspired (neural computation), behavioral (fuzzy rules), or evolutionary (search over candidate solutions). In this chapter we focus on the statistical, data-driven strand: supervised learning.

In this sense, supervised learning is about prediction and inference: given evidence $\mathbf{x}$, estimate an output y that you can act on or audit. Other modeling goals exist (summarizing structure, compressing representations, discovering clusters), but supervised learning is the cleanest place to learn the mechanics of fitting models, comparing alternatives, and checking whether your success is real or just memorization.

Supervised learning begins with three commitments: pick a functional form

that can plausibly approximate the mapping, collect paired examples of inputs and outputs, and define a quantitative measure of "how wrong" a prediction is. Once those are in place, training becomes possible: we adjust the model parameters so the predictions align with the observed outputs on the examples we have.

The word *fitting* is meant literally. In classical curve fitting—and in practical settings like sensor calibration—we choose parameters so a predicted curve (or surface) passes near measured points. Keep the camera thread from Chapter 1 in mind: a camera system is useful because it can predict something actionable from what it senses. A simple example is exposure calibration: we collect scenes with known reference targets, measure raw sensor readouts $\mathbf{x}$, and learn parameters that map those readouts to a correction y so the system produces consistent brightness across conditions. The same pattern repeats at higher levels (object detection scores, tracking signals, alert decisions): the details change, but the core act is the same—use paired input/output examples to fit parameters that make predictions reliable.

This chapter builds the supervised-learning toolkit around that central act. We start by making the pieces explicit (data, models, and losses). Then we show what training is doing when it succeeds, and what it looks like when it fails (underfitting vs. overfitting). After that we formalize the standard objective (ERM) and the main "anti-memorization" tools (regularization and validation). Finally, we work through linear regression as the first fully transparent case study where you can see the entire pipeline end to end.

Design motif

Data → model → objective → audit. This workflow shows up repeatedly in later chapters, even when the models become deeper and the optimization less forgiving.

Risk & audit

- **Leakage:** avoid split mistakes (duplicates, near-duplicates, time leakage) that inflate validation accuracy.
- **Metric mismatch:** align the loss you optimize with the metric you report (and the decision you must make).
- **Overfitting signals:** track training vs. validation curves and use learning curves to diagnose data hunger vs. excess capacity.
- **Distribution shift:** audit performance by slice (population, device, lighting, region) rather than relying on one aggregate score.
- **Calibration:** check reliability when probabilities drive actions (thresholds, alerts, resource allocation).

A concrete toy task

To keep the discussion grounded, we will repeatedly sketch one small binary classification problem: a two-moons toy with a standard train/validation/test split. It is deliberately simple, but it is rich enough to reveal the recurring failure modes (memorization, metric mismatch, split leakage) and the recurring remedies (regularization, validation, and diagnostics).

Before we turn the "fitting" story into equations, let us fix the handful of symbols we will reuse for several pages. The goal is not to introduce new notation, but to keep the derivations readable while the ideas are still new.

- Data $X \in \mathbb{R}^{N \times d}$ with rows $\mathbf{x}_i^\top$; targets $y_i \in \mathbb{R}$ for regression and $y_i \in \{0, 1\}$ for binary classification. The affine map $y_{\pm 1} = 2y - 1$ switches to $\{-1, +1\}$ when margin-based expressions are convenient.
- Parameters θ (model-specific), weights $\mathbf{w} \in \mathbb{R}^d$; predictions carry hats: $\hat{y}_i = h_\theta(\mathbf{x}_i)$, $\hat{\mathbf{y}} = X\mathbf{w}$.
- The loss $\ell(\hat{y}, y)$ is the teacher's grading rubric; the objective aggregates losses over data and adds regularization. Parameters are learned from data; hyperparameters (e.g., λ in regularization) are chosen by validation.
- Noise uses ε; residuals use $e = y - \hat{y}$. Vectors are bold lowercase, matrices

bold uppercase; scalars are italic.

- Bias absorption (when used): augmented feature $\tilde{\mathbf{x}} = [\mathbf{x}; 1]$ with corresponding augmented weights.

3.1 Problem Setup and Notation

We observe a dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ drawn i.i.d. from an unknown distribution $\mathcal{P}$ on the input–output space $\mathcal{X} \times \mathcal{Y}$. A hypothesis (model) $h_\theta : \mathcal{X} \to \mathcal{Y}$ with parameters θ produces predictions $\hat{y}_i = h_\theta(\mathbf{x}_i)$. A pointwise loss function $\ell(\hat{y}, y)$ measures the penalty incurred by predicting $\hat{y}$ when the true label is y.

The *population risk* and *empirical risk* associated with h_θ are

$$R(h_\theta) = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{P}}\big[\ell\big(h_\theta(\mathbf{x}), y\big)\big], \tag{3.1}$$

$$\hat{R}_N(h_\theta) = \frac{1}{N}\sum_{i=1}^{N} \ell\big(h_\theta(\mathbf{x}_i), y_i\big). \tag{3.2}$$

Because $\mathcal{P}$ is unknown, learning algorithms minimize empirical proxies of $R(h_\theta)$. This is the formal version of the "educated guess" idea: we posit a model family h_θ, then use data to choose parameter values that make its predictions behave like the measured input–output pairs. In practice we do this on a *training set* (the data used to fit parameters), and we reserve held-out data to check whether the fitted model is trustworthy; Section 3.6 makes these evaluation protocols precise.

3.2 Fitting, Overfitting, and Underfitting

Fitting is the act of choosing parameters θ so a model's predictions match observed data. Concretely, we pick a loss ℓ, evaluate it on examples $(\mathbf{x}_i, y_i)$, and use an optimization method to search for parameters that make the aggregate loss small. This is what practitioners usually mean by *training*.

It helps to picture training as repeated adjustment under feedback. You make a prediction, measure the mistake with a loss, update parameters to reduce that mistake, and repeat. In sensor calibration, this feels familiar: if your measured output is consistently off, you change a gain or offset; if it is noisy, you adjust how aggressively you trust any one reading. Supervised learning packages that intuition into a general recipe that can be reused across problems.

The goal is not to "fit the training set" as an end in itself. A good fit is one that holds up on new data: the fitted model should behave sensibly on inputs it has not seen. When fitting fails, it tends to fail in one of two recognizable ways.

Underfitting. The model family is too rigid for the task, the features do not contain enough information, or the optimization did not do its job. This is the student who cannot solve the practice problems before the exam: the mismatch is obvious even on the training set. The remedy is to change the representation or the hypothesis class, improve the data, or fix the optimization.

Overfitting. The model is flexible enough to match the training set by memorizing its quirks. This is the student who memorizes the worked examples so well that a small twist on the exam causes failure. Overfitting can look like success until you test on held-out data.

What we aim for. We want a well-fitted model: low training error and comparable validation/test error. The tools below are designed to keep that distinction visible: objectives (losses and regularizers), validation protocols (splits and cross-validation), and diagnostics (learning curves and bias–variance reasoning).

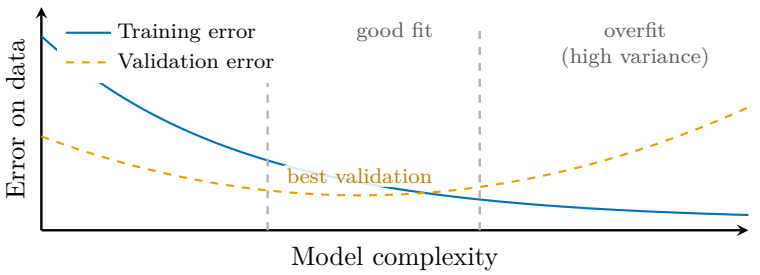

Figure 3: Schematic: Underfitting and overfitting as a function of model complexity. Training error typically decreases with complexity, while validation error often has a U-shape. Regularization and model selection aim to operate near the minimum of the validation curve.

3.3 Empirical Risk Minimization and Regularization

To make the informal idea of "fitting" mathematically precise, we choose an objective and minimize it. The supervised-learning baseline is *empirical risk*

minimization: minimize the average loss on the training data.

This is the first place where the chapter's opening promises become concrete. If we do not know the true input–output law, we still need a disciplined way to compare candidate models and to say whether one parameter choice is better than another. The loss plays the role of the teacher's rubric, and ERM is the simplest way to aggregate that rubric over many examples: instead of arguing about one example at a time, we ask for parameters that perform well on average across the dataset.

The *empirical risk minimizer* (ERM) selects

$$\hat{\theta}_{\mathrm{ERM}} = \arg\min_{\theta} \ \hat{R}_N(h_\theta). \tag{3.3}$$

To mitigate overfitting, we often add a regularizer $\Omega(\theta)$ with strength $\lambda \geq 0$:

$$\hat{\theta}_\lambda = \arg\min_{\theta} \ \hat{R}_N(h_\theta) + \lambda\,\Omega(\theta), \qquad \Omega(\theta) \in \{\|\theta\|_2^2,\ \|\theta\|_1, \ldots\}. \tag{3.4}$$

Regularization is not an arbitrary penalty. It is the mathematical version of a teaching move: if a student can memorize every worked example, you change the exercises so memorization is less effective and understanding is rewarded. Regularization plays the same role. It makes some parameter settings expensive, which pushes learning toward explanations that generalize better.

In supervised learning, this matters because a model can drive training loss down in ways that do not survive contact with new data. Regularization is one of the main tools we use to "push back" against memorization: we still fit the data, but we also express a preference for solutions that are stable, simple, or structured in ways that match the problem.

Ridge and lasso. Two penalties show up so often that they have become part of the basic vocabulary:

- **Ridge (L2)** adds $\|\theta\|_2^2$, which shrinks weights smoothly and stabilizes solutions when features are correlated.
- **Lasso (L1)** adds $\|\theta\|_1$, which tends to set some weights to exactly zero, yielding sparse models and a form of feature selection.

The difference is easiest to remember geometrically: L2 has round level sets,

while L1 has corners, and corners create exact zeros.

Regularization: L1/L2 and scaling

- **Why regularize?** Flexible models can fit training data by effectively memorizing idiosyncrasies (noise, quirks of the sample) rather than capturing stable structure. Regularization makes such memorization expensive and rewards explanations that survive on held-out data.
- **L2 (ridge)** shrinks weights smoothly, is rotationally invariant, and works well when features are dense and correlated.
- **L1 (lasso)** promotes sparsity, effectively performing feature selection when many coefficients should be zero.
- **Why the names?** "Ridge" refers to the ridge-like valleys that appear in least-squares objectives under multicollinearity; the L2 penalty lifts the valley floor and stabilizes the solution. "LASSO" is an acronym for *Least Absolute Shrinkage and Selection Operator*.
- **Why L1 vs. L2 feels different:** the L2 penalty discourages large coefficients but rarely drives them exactly to zero, while the L1 penalty creates corners in the geometry that tend to set some coefficients to exactly zero.
- **Standardization** (zero mean, unit variance) is essential before applying L1/L2/elastic-net so the penalty treats all dimensions comparably.
- With an intercept term, centering y makes the algebra cleaner; ridge and lasso still apply directly once features are scaled.

3.4 Elastic-net paths and cross-validation

Pure L1 or L2 penalties rarely dominate modern workflows; the *elastic net* mixes them to balance sparsity and stability:

$$\hat{\theta}_{\alpha,\lambda} = \arg\min_{\theta} \hat{R}_N(h_\theta) + \lambda\Big(\alpha\|\theta\|_1 + \tfrac{1-\alpha}{2}\|\theta\|_2^2\Big), \qquad \alpha \in [0,1]. \tag{3.5}$$

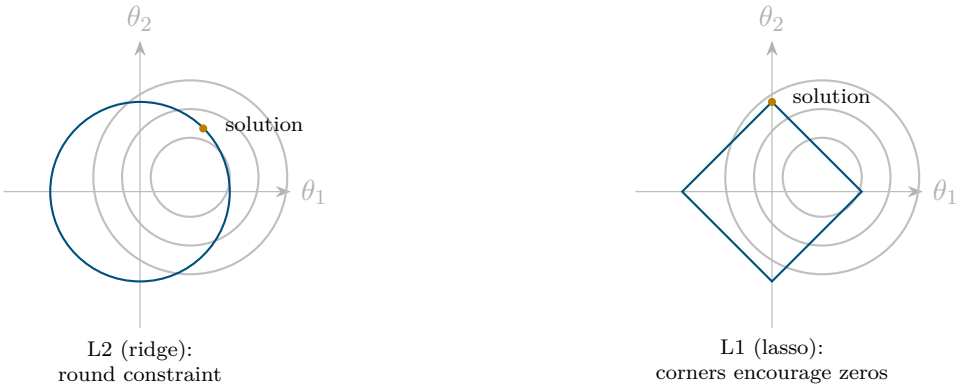

Figure 4: Schematic: Why L1 promotes sparsity. Minimizing loss subject to an L2 constraint tends to hit a smooth boundary; an L1 constraint has corners aligned with coordinate axes, so tangency often occurs at a point where some coordinates are exactly zero.

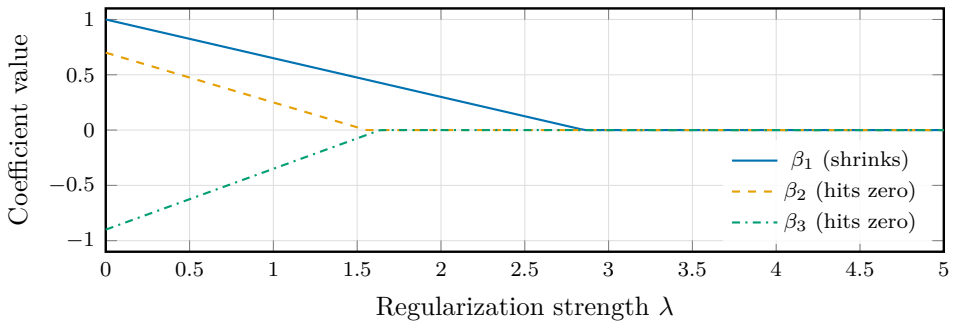

Figure 5: Schematic: A typical lasso path as the regularization strength increases. Coefficients shrink, and some become exactly zero, yielding sparse models.

Setting $\alpha = 1$ recovers the lasso, $\alpha = 0$ yields ridge, and intermediate values trace a solution path that tends to group correlated features while still pruning irrelevant ones. In practice we standardize the features once, draw a logarithmic grid of λ values, and run K-fold cross-validation for each pair (α, λ). The "one-standard-error" rule selects the largest λ whose validation error is within one standard error of the minimum. It gives a stable operating point and avoids over-interpreting tiny validation differences.

3.5 Common Loss Functions

Loss functions make the teacher signal quantitative: they decide what counts as a small mistake, what counts as a large one, and which kinds of errors matter most. For binary classification with labels $y \in \{-1, +1\}$ and margin $z = y\, f(\mathbf{x})$, two standard losses are

$$\ell_{\text{hinge}}(y, z) = \max\bigl(0, 1 - z\bigr), \qquad \ell_{\text{logistic}}(y, z) = \log\bigl(1 + e^{-z}\bigr). \tag{3.6}$$

Here $y \in \{-1, +1\}$; when labels are instead coded as $y \in \{0, 1\}$ (common in probability-of-class formulations), the margin expression uses $y_{\pm 1} = 2y - 1$ to map between codings. Figure 6 visualizes these curves together with the squared hinge so you can match the algebra to the margin geometry. For regression with residual $e = y - \hat{y}$, we frequently use

$$\ell_{\text{sq}}(e) = \tfrac{1}{2}e^2, \qquad \ell_{\text{abs}}(e) = |e|. \tag{3.7}$$

The Huber loss interpolates between these: it is quadratic when $|e| \le \delta$ and linear beyond that threshold (here the plot uses $\delta = 1$), reducing sensitivity to outliers while remaining smooth around the origin.

Table 2: Schematic: Common losses and typical use (reference for Chapters 3 to 5).

Loss	**Convex?**	**Typical use**
Squared error $\frac{1}{2}e^2$	Yes	Regression when Gaussian noise is plausible; differentiable everywhere.
Absolute error $\lvert e\rvert$	Yes	Robust regression with Laplacian noise assumptions; non-differentiable at 0.
Huber (quadratic $\to$ linear)	Yes	Regression when moderate outliers are present; smooth near zero.
Logistic (binary cross-entropy)	Yes	Probabilistic classification; pairs naturally with sigmoid.
Hinge / squared hinge	Yes	Margin-based classifiers (SVMs, large-margin perceptrons).

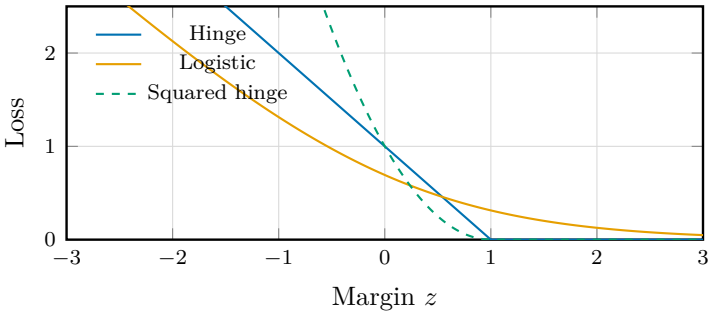

Figure 6: Schematic: Classification losses as functions of the signed margin z.

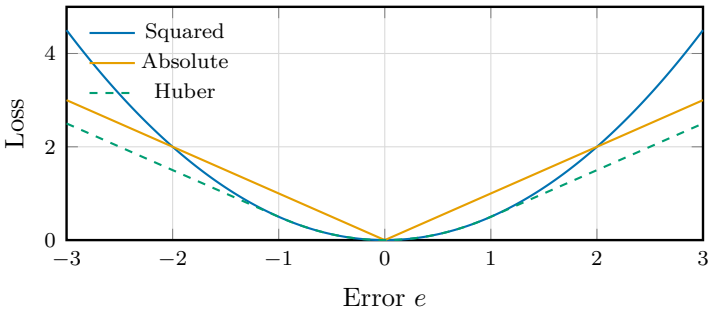

Figure 7: Schematic: Regression losses versus prediction error. The Huber loss transitions from quadratic to linear to reduce sensitivity to outliers.

3.6 Model Selection, Splits, and Learning Curves

Up to this point, we have defined what it means to fit: choose a model family, pick an objective (loss plus any regularizer), and tune parameters to reduce that objective on observed examples. The next question is how to choose among competing model families, hyperparameters, and training procedures without fooling ourselves. Model selection is the discipline of making those choices using validation data, while keeping one final dataset split untouched so that the reported performance remains honest.

In other words, this is where the chapter's "audit" step becomes operational: we decide what to trust by checking performance on data the model has not been allowed to fit.

Practical workflows allocate data into training, validation, and test portions. Training data are used to fit parameters; validation data guide choices such as hyperparameters and model families; and the test set provides an unbiased audit once those choices are fixed. The key habit is the separation of roles: training is where you allow the model to "learn" (and potentially overfit), validation is where you decide what kind of learning you trust, and the test set is the final audit.

Proper scoring rules and calibration

- **Log loss (cross-entropy)** and the **Brier score** are *proper* scoring rules: in expectation, they are minimized by predicting the true class probability.
- **Brier** is squared error in probability space; it penalizes confident mistakes less harshly than log loss and is often paired with reliability diagrams.
- **Log loss** heavily punishes overconfident errors (loss $\to \infty$ as predicted probability $\to 0$ on the true class), so it is a natural objective when probabilities will be thresholded downstream.
- **Practical tip:** train with log loss, but monitor both log loss and Brier score on validation data to catch calibration issues early.

To make the workflow concrete, Figure 9 summarizes the standard ERM pipeline from dataset to model selection.

Learning curves plot training and validation error against the number of training examples, revealing underfitting or overfitting regimes.

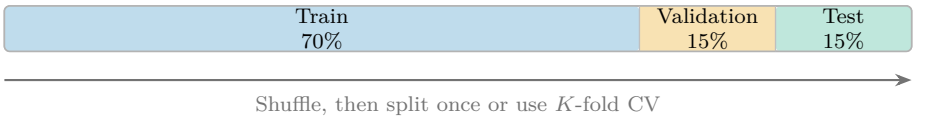

Figure 8: Schematic: Dataset partitioning into training, validation, and test segments. Any resampling scheme should preserve disjoint evaluation data; when classes are imbalanced, shuffle within strata so each split reflects the overall class mix.

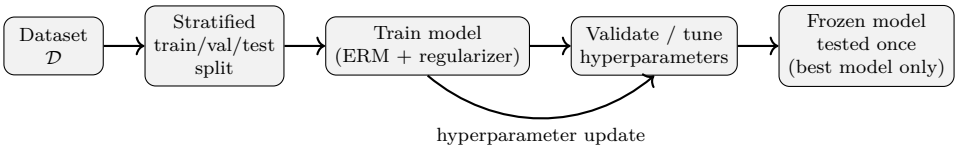

Figure 9: Schematic: Mini ERM pipeline (split once, iterate train/validate, then test only the best model on the held-out set).

Data-leakage checklist

- Split data before any preprocessing or feature selection.
- Fit scalers/imputers/dimensionality-reduction transforms on the training fold only; reuse fitted parameters on validation/test (or within each CV fold via pipelines).
- Respect temporal order for time-series; avoid target/future-derived features.
- Wrap preprocessing + model in a pipeline for cross-validation so transformers refit inside each fold.

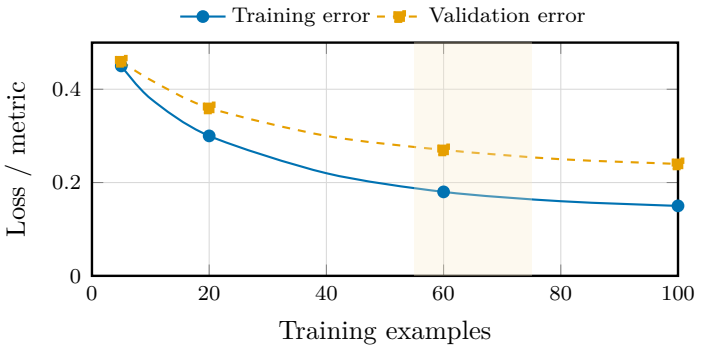

Figure 10: Schematic: Learning curves reveal under/overfitting: the validation curve flattens while additional data continue to decrease training error only marginally. A shaded patience window marks when early stopping would halt if no validation improvement occurs.

Bias–variance at a glance

- **High bias (underfit):** train and validation errors both plateau high and together; add capacity/features or reduce regularization.
- **High variance (overfit):** train error low, validation error high/diverging; add data, strengthen regularization, or use early stopping.
- **Well fit:** train/validation track closely and decrease or level off at low error; further gains require better data or priors.

Learning curves explain *why* the train/validation split is useful: they show whether more data, more capacity, or more regularization is the lever that actually moves the validation error. Once you can read these curves, a natural next question is what happens as we scale up data and model size. The aside below summarizes two modern empirical patterns that are best treated as guidance, not as a recipe.

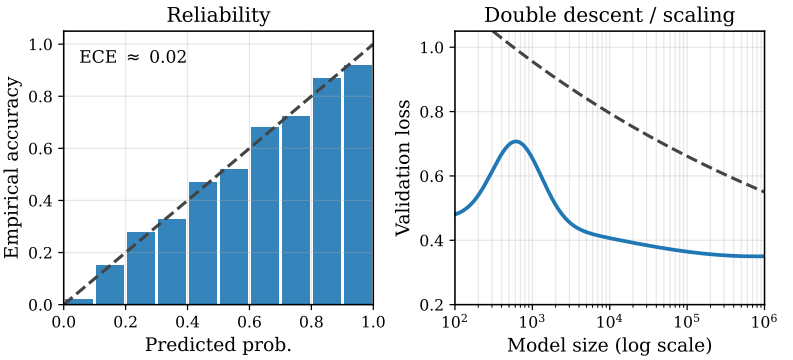

Figure 11: Schematic: Calibration and capacity diagnostics. Left: reliability diagram with binned predicted probabilities vs. empirical accuracy; Expected Calibration Error (ECE) measures deviation from the diagonal. Right: illustrative double-descent risk vs. model size (log-scale on the x-axis); dashed line sketches a scaling-law trend used to choose capacity/regularization.

Aside: scaling laws and double descent

The simplest story is the classical bias–variance picture: as model capacity grows, training error falls, and validation error often has a U-shape. In modern overparameterized models, that picture can be incomplete. You may see *double descent*: after the classical U-shape, error can decrease again once model size exceeds the interpolation threshold (Belkin et al., 2019). You may also hear *scaling laws*: in some regimes, validation loss decreases roughly as a power law of compute, data, and model size (Kaplan et al., 2020; Hoffmann et al., 2022).

Treat both as diagnostics rather than guarantees. Use them to decide whether to collect more data, shrink or expand a model, or regularize more aggressively, but still make final choices by comparing validation curves. Do not chase the interpolation peak as a goal.

Regularization trades model complexity for generalization; Figure 12 depicts the effect of ridge penalties on the weight norm.

3.7 Linear regression: a first full case study

Up to this point, the supervised-learning pipeline has been described in abstract terms: a dataset, a hypothesis class, an objective, and an audit. Linear regres-

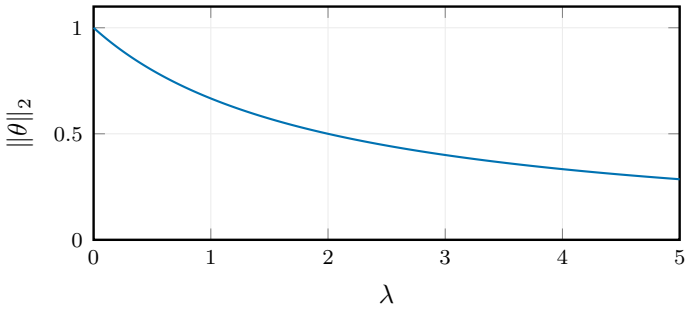

Figure 12: Schematic: Ridge regularization shrinks parameter norms as the penalty strength increases.

sion is where those pieces become concrete enough that you can see every moving part at once.

A useful habit, before you commit to any model family, is to ask whether there is any signal to model in the first place. If the relationship were deterministic (like Celsius $\leftrightarrow$ Fahrenheit), there would be nothing to learn. In supervised learning we assume the relationship is statistical: the same input can map to different outputs because of noise, missing variables, or genuine uncertainty. For simple problems, a scatter plot and a correlation coefficient can reveal whether a linear trend is even plausible. For high-dimensional data, analogous sanity checks (feature scaling, collinearity) help you decide whether linear regression is a sensible starting point or merely a baseline.

In this section, the coefficients $\boldsymbol{\beta}$ are the knobs we turn. "Learning" means using training pairs $(X, \mathbf{y})$ to estimate $\boldsymbol{\beta}$ so predictions match observed targets as well as they can under the chosen objective. Because least squares is a convex, closed-form problem, repeating the fit with the same data returns the same solution (up to numerical tolerance). That transparency is exactly why linear regression is worth treating carefully: it makes the ideas of losses, optimization, regularization, and validation feel concrete before we move on to models where the same pipeline is less visible.

Model. Given inputs $\mathbf{x}_i \in \mathbb{R}^d$ and continuous targets $y_i \in \mathbb{R}$, the linear model predicts

$$\hat{\mathbf{y}} = X\boldsymbol{\beta}, \qquad X \in \mathbb{R}^{N \times d}. \tag{3.8}$$

Equivalently, $\hat{y}_i = \mathbf{x}_i^\top \boldsymbol{\beta}$ for each data point. The vector $\boldsymbol{\beta}$ is the set of adjustable parameters: *fitting* the model means choosing $\boldsymbol{\beta}$ so that predictions align with observed outputs. The residual $\mathbf{e} = \mathbf{y} - \hat{\mathbf{y}}$ captures what the model fails to explain on the data at hand.

A noise model (why squared error shows up). A common way to formalize "measurement scatter" is to write

$$y_i = \mathbf{x}_i^\top \boldsymbol{\beta} + \varepsilon_i, \qquad \varepsilon_i \sim \mathcal{N}(0, \sigma^2), \tag{3.9}$$

where ε_i is observation noise (sensor noise, unmodeled effects, annotation noise, etc.). Under this assumption,

$$p(y_i \mid \mathbf{x}_i, \boldsymbol{\beta}) = \mathcal{N}(\mathbf{x}_i^\top \boldsymbol{\beta}, \sigma^2), \tag{3.10}$$

and (assuming i.i.d. observations) the likelihood factorizes:

$$p(\mathbf{y} \mid X, \boldsymbol{\beta}) = \prod_{i=1}^{N} p(y_i \mid \mathbf{x}_i, \boldsymbol{\beta}). \tag{3.11}$$

Maximizing the (log) likelihood is equivalent to minimizing the negative log-likelihood, and for Gaussian noise that becomes (up to constants and a scale factor $1/(2\sigma^2)$) the familiar sum of squared errors:

$$-\log p(\mathbf{y} \mid X, \boldsymbol{\beta}) = \frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - \mathbf{x}_i^\top \boldsymbol{\beta})^2 + \text{const.} \tag{3.12}$$

This is the simplest example of a recurring theme: if you propose an "educated guess" model for how data are generated, training often becomes "minimize a loss".

Objective. To fit $\boldsymbol{\beta}$, we need a grading rubric. Squared error is the standard starting point because it is smooth and strongly penalizes large mistakes:

$$L(\boldsymbol{\beta}) = \tfrac{1}{2} \|\mathbf{y} - X\boldsymbol{\beta}\|_2^2. \tag{3.13}$$

The gradient is simple,

$$\nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}) = X^\top (X\boldsymbol{\beta} - \mathbf{y}), \tag{3.14}$$

so gradient descent is explicit: $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \eta X^\top (X\boldsymbol{\beta} - \mathbf{y})$. This same loop reappears later when the model is no longer linear and the loss is no longer quadratic.

Closed form and geometry. Least squares is convex and satisfies the normal equations:

$$X^\top X\, \hat{\boldsymbol{\beta}} = X^\top \mathbf{y}. \tag{3.15}$$

When $X^\top X$ is invertible, the solution can be written explicitly as

$$\hat{\boldsymbol{\beta}} = (X^\top X)^{-1} X^\top \mathbf{y}. \tag{3.16}$$

Geometrically, the prediction $\hat{\mathbf{y}}$ is the orthogonal projection of $\mathbf{y}$ onto the column space of X. In code, solve the linear system (QR/SVD) rather than forming $(X^\top X)^{-1}$ explicitly; collinearity can make $X^\top X$ poorly conditioned even when the mathematics is correct.

Where overfitting enters. With raw features, a linear model may underfit; with aggressive feature expansions (polynomials, splines, kernels, learned features), the same least-squares machinery can overfit. This is where the earlier tools matter. Regularization (ridge, lasso, elastic net) makes memorization harder; validation selects hyperparameters; learning curves diagnose whether error is limited by bias, variance, or data.

Ridge and lasso in one line. Ridge adds $\lambda \|\boldsymbol{\beta}\|_2^2$ to the objective, shrinking coefficients and stabilizing solutions when features are correlated; lasso uses $\|\boldsymbol{\beta}\|_1$ and tends to drive some coefficients to exactly zero. The ridge shrinkage behavior is visualized in Figure 12.

The discipline of supervised learning is reusable across models: define a dataset and a hypothesis class, choose an objective (loss plus regularizer), optimize it, and then audit generalization with clean train/validation/test separation. In Chapter 4, we apply this toolkit to classification, where the loss becomes a

Bernoulli negative log-likelihood (cross-entropy) and the evaluation tools expand (confusion matrices, ROC/PR curves, and calibration).

Key takeaways

- Supervised learning chooses a hypothesis class and fits parameters by minimizing empirical risk, then audits generalization on held-out data.
- Overfitting is a training success but a deployment failure; regularization and validation protocols are the practical defenses.
- Learning curves and bias–variance reasoning are diagnostics: they help decide whether to add data, change capacity, or adjust regularization.

Exercises and lab ideas

- Implement linear regression with ridge regularization using both (i) a closed-form solve (QR/SVD) and (ii) gradient descent; compare validation curves as λ varies.
- Create a controlled overfitting experiment: increase polynomial feature degree or add noisy features, then use learning curves (Figure 10) to diagnose bias vs. variance and decide how much regularization is needed.
- Demonstrate a leakage failure mode by fitting preprocessing on the full dataset (incorrect) versus on the training split only (correct); report the difference in test error.

Where we head next. In Chapter 4, we shift from continuous targets to discrete labels: the model predicts class probabilities, the loss becomes cross-entropy, and the evaluation toolkit expands (confusion matrices, ROC/PR curves, and threshold selection).

References. Full citations for works mentioned in this chapter appear in the book-wide bibliography.

4 Classification and Logistic Regression

Learning Outcomes

After this chapter, you should be able to:

- Derive the logistic log-likelihood and its gradient.
- Explain the NLL (cross-entropy) connection and convexity.
- Extend to softmax regression for multiclass problems.

Chapter 2 illustrated one view of intelligent behavior as transformation search with explicit goal tests, while Chapter 3 introduced the data-driven view: represent a task with data, choose a hypothesis class, minimize empirical risk under regularization, and audit performance on held-out data. This chapter extends that toolkit from regression to classification. The roadmap in Figure 1 places this chapter on the core supervised path.

Design motif

Keep the workflow, swap the likelihood: logistic regression keeps the linear score but changes the probabilistic model (Bernoulli likelihood) and the loss (negative log-likelihood, i.e., cross-entropy).

Risk & audit

- **Threshold choice:** a high AUC can still yield a poor operating point; pick thresholds using a validation objective that matches the cost.
- **Class imbalance:** report PR curves (or per-class metrics) when positives are rare; audit confusion matrices, not just accuracy.
- **Probability quality:** logits can be miscalibrated; use reliability diagrams/ECE when probabilities are consumed downstream.
- **Feature shortcuts:** strong apparent accuracy can come from spurious correlates; sanity-check with slices and perturbations.
- **Regularization:** tune λ with held-out data; do not report the best test result after repeated tuning.

Worked example: a toy decision boundary

On a two-moons binary classification toy, the ideas in this chapter have a visible effect:

- Logistic regression gives a calibrated probabilistic baseline (linear score + sigmoid) that is easy to diagnose.
- The nonlinearity we will need later is *not* in the optimizer but in the representation: Chapter 6 introduces multilayer features and Chapter 7 supplies the training engine that lets those features improve the decision boundary.

4.1 From regression to classification

Linear regression models a continuous target y and, under a Gaussian noise model, yields a closed-form solution via the normal equations (Chapter 3, Section 3.7). Classification changes the output space: y is a discrete label, and the model predicts class probabilities rather than raw responses. The ERM pipeline remains the same, but the loss becomes the negative log-likelihood (binary cross-entropy) and optimization is typically iterative.

Throughout this chapter we use $y \in \{0, 1\}$ by default, switching to $y_{\pm 1} = 2y - 1$ only when margin-based expressions are convenient. Predictions carry hats (e.g., $\hat{y}$), ε denotes noise, and $e = y - \hat{y}$ denotes residuals. For a refresher on data splits, learning curves, and the bias–variance vocabulary, see Chapter 3; here we focus on the logistic-specific modeling and diagnostics.

4.2 Classification problem statement

In this chapter, we shift our attention to a fundamentally different type of problem: *classification*. Unlike regression, where the output y is continuous, classification predicts a *discrete label*. We will start with the binary case $y \in \{0, 1\}$, where the goal is to estimate a probability

$$\pi(\mathbf{x}) = P(y = 1 \mid \mathbf{x}),$$

and then produce a decision by thresholding $\pi(\mathbf{x})$ (or by comparing class probabilities when there are more than two classes). For multiclass problems the label

belongs to one of K classes,

$$y \in \{c_1, c_2, \ldots, c_K\},$$

and the goal is to estimate $P(y = c_k \mid \mathbf{x})$ for each k; we return to the softmax extension later in the chapter.

4.3 Bayes Optimal Classifier

A fundamental result in statistical pattern recognition is that the *Bayes classifier* is the optimal classifier in terms of minimizing the expected classification error. The Bayes classifier assigns $\mathbf{x}$ to the class:

$$\hat{y} = \arg \max_{c_k \in \{c_1, \ldots, c_K\}} P(y = c_k \mid \mathbf{x}).$$

Using Bayes' theorem, the posterior probability can be expressed as

$$P(y = c_k \mid \mathbf{x}) = \frac{P(\mathbf{x} \mid y = c_k) P(y = c_k)}{P(\mathbf{x})}. \tag{4.1}$$

Here

- $P(\mathbf{x} \mid y = c_k)$ is the *class-conditional likelihood*,
- $P(y = c_k)$ is the *prior probability* of class c_k,
- $P(\mathbf{x}) = \sum_{j=1}^{K} P(\mathbf{x} \mid y = c_j) P(y = c_j)$ is the marginal likelihood of the input.

Equation (4.1) provides a principled way to compute the posterior probabilities, and thus the optimal classification rule.

Challenges in Practice Despite its theoretical appeal, the Bayes classifier is rarely used directly in practice because:

- The class-conditional densities $P(\mathbf{x} \mid y = c_k)$ are typically unknown.
- The prior probabilities $P(y = c_k)$ may also be unknown or difficult to estimate accurately.
- Estimating these distributions nonparametrically or parametrically can be challenging, especially in high-dimensional spaces.

Consequently, practical classification methods often rely on approximations or alternative formulations.

Running example: a two-cluster dataset To keep the discussion concrete, we reuse a small toy dataset in Figure 13 consisting of two Gaussian clusters. Under a simple generative assumption (equal covariances and similar priors), Figure 14 visualizes the Bayes-optimal decision boundary: it is linear in this LDA setting, while unequal covariances yield a quadratic boundary. This running example will anchor the geometric intuition (what a decision boundary looks like) before we turn to discriminative models that learn $\pi(\mathbf{x})$ directly.

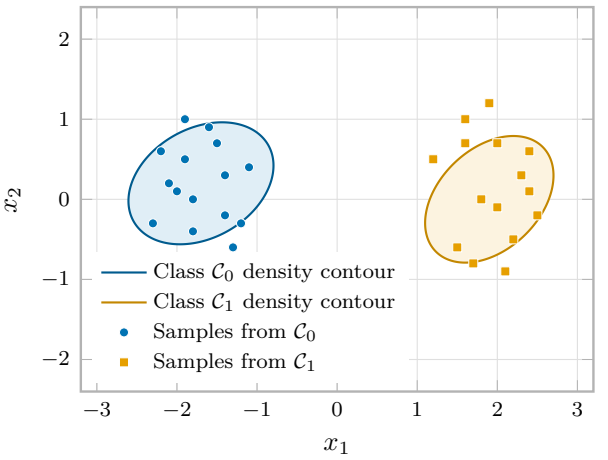

Figure 13: Schematic: Synthetic binary dataset built from two anisotropic Gaussian clusters; shaded ellipses hint at the underlying density while the scattered samples are reused throughout the running examples.

Naive Bayes Approximation One classical workaround is the Naive Bayes classifier, which assumes that the components of $\mathbf{x}$ are conditionally independent given the class label. Under this assumption,

$$P(\mathbf{x} \mid y = c_k) = \prod_{j=1}^{p} P(x_j \mid y = c_k),$$

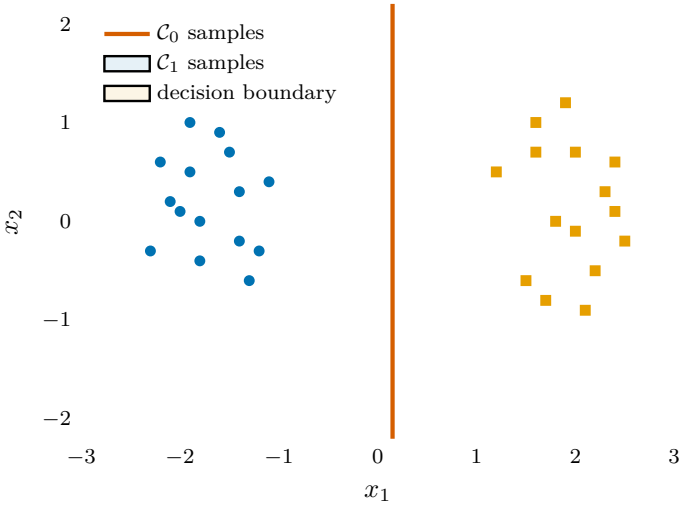

Figure 14: Schematic: Bayes-optimal boundary for two Gaussian classes with equal covariances and similar priors (LDA setting), which yields a linear separator. Unequal covariances produce a quadratic boundary. We place the boundary near the equal-posterior line (vertical, pink); left/right regions correspond to predicted classes R0 and R1.

making the computation and estimation of the likelihood tractable. It is important to remember that this factorization is justified only under the conditional independence assumption; when the features are strongly correlated, Naive Bayes can suffer because the assumption is violated.

4.4 Logistic Regression: A Probabilistic Discriminative Model

One widely used approach to classification, especially for binary problems, is *logistic regression*. Logistic regression models the posterior probability $P(y = 1 \mid \mathbf{x})$ directly as a function of $\mathbf{x}$, without explicitly modeling the class-conditional densities.

Logistic regression at a glance

Objective: Minimize binary cross-entropy (negative log-likelihood) between true labels $y \in \{0, 1\}$ and predicted probabilities $\hat{p} = \sigma(\boldsymbol{\beta}^\top \tilde{\mathbf{x}})$.
Key hyperparameters: Regularization type/strength (L2 or L1, penalty λ; many libraries use $C = 1/\lambda$), feature scaling, optimization settings (step size, iterations).
Defaults: Standardize features; use L2 regularization with moderate strength; start with a 0.5 decision threshold and adjust only if class costs are asymmetric.
Common pitfalls: Strong collinearity between features, severe class imbalance, and uncalibrated probability outputs if the model is over-regularized or trained on a biased sample.

Binary Classification Setup Consider the binary classification problem where $y \in \{0, 1\}$. The goal is to model the probability that the output is class 1 given the input $\mathbf{x}$: $P(y = 1 \mid \mathbf{x}) = \pi(\mathbf{x})$.

Linear Model for the Log-Odds Logistic regression assumes that the *log-odds* (also called the *logit*) of the positive class is a linear function of the input features. Introducing the augmented feature vector $\tilde{\mathbf{x}} = [1, x_1, \ldots, x_p]^\top$ and parameter vector $\boldsymbol{\beta} = [\beta_0, \beta_1, \ldots, \beta_p]^\top$, we write

$$\log \frac{\pi(\mathbf{x})}{1 - \pi(\mathbf{x})} = \boldsymbol{\beta}^\top \tilde{\mathbf{x}}. \tag{4.2}$$

This implies that the posterior probability $\pi(\mathbf{x})$ can be written as the *logistic sigmoid* function applied to the linear predictor:

$$\pi(\mathbf{x}) = \frac{1}{1 + \exp\left(-\boldsymbol{\beta}^\top \tilde{\mathbf{x}}\right)}. \tag{4.3}$$

Author's note: why "logistic" and why "regression"?

The name *logistic* comes from the logistic (sigmoid) link in Equation (4.3), which maps a real-valued score to a probability in $[0, 1]$. The word *regression* reflects what we model linearly: the *log-odds* (logit) in Equation (4.2) is a linear function of the features. The model itself outputs a continuous probability; we turn that probability into a class label by thresholding (or comparing class probabilities in the multiclass extension).

4.4.1 Likelihood, loss, and gradient

For data $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ with $y_i \in \{0, 1\}$, define $p_i = \pi(\mathbf{x}_i) = \sigma(\boldsymbol{\beta}^\top \tilde{\mathbf{x}}_i)$. Under a Bernoulli model, the likelihood factorizes as

$$p(\mathbf{y} \mid X, \boldsymbol{\beta}) = \prod_{i=1}^{N} p_i^{y_i} (1 - p_i)^{1 - y_i}, \tag{4.4}$$

so the log-likelihood is

$$\log p(\mathbf{y} \mid X, \boldsymbol{\beta}) = \sum_{i=1}^{N} \Big(y_i \log p_i + (1 - y_i) \log(1 - p_i) \Big). \tag{4.5}$$

Maximizing Equation (4.5) is equivalent to minimizing the negative log-likelihood (binary cross-entropy). With the design matrix $X = [\tilde{\mathbf{x}}_1^\top; \ldots; \tilde{\mathbf{x}}_N^\top]$ and vector $\mathbf{p} = (p_1, \ldots, p_N)^\top$, the gradient of the negative log-likelihood is

$$\nabla_{\boldsymbol{\beta}} \Big(- \log p(\mathbf{y} \mid X, \boldsymbol{\beta}) \Big) = X^\top (\mathbf{p} - \mathbf{y}). \tag{4.6}$$

The Hessian has the form $X^\top W X$ where $W = \operatorname{diag}(p_i(1 - p_i)) \succeq 0$, which makes the objective convex and explains why second-order methods work well for moderate feature dimensions.

Optimization geometry (why iterative solvers) Unlike linear regression, logistic regression does not have a closed-form solution for $\boldsymbol{\beta}$, even though the objective is convex. In practice we therefore rely on iterative solvers (gradient methods, quasi-Newton methods, or Newton/IRLS in moderate dimensions). Figure 16 is a convex quadratic toy that reminds us what an optimization trajec-

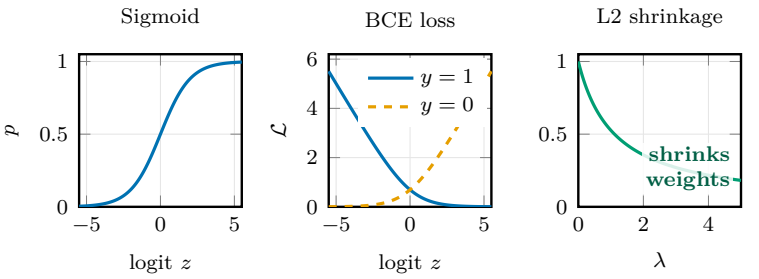

Figure 15: Schematic: The sigmoid maps logits to probabilities (left). The binary cross-entropy (negative log-likelihood) penalizes confident wrong predictions sharply (middle). Regularization typically shrinks parameter norms as the penalty strength increases (right).

tory looks like when we minimize a smooth objective: step size and conditioning shape how quickly iterates contract toward the minimizer.

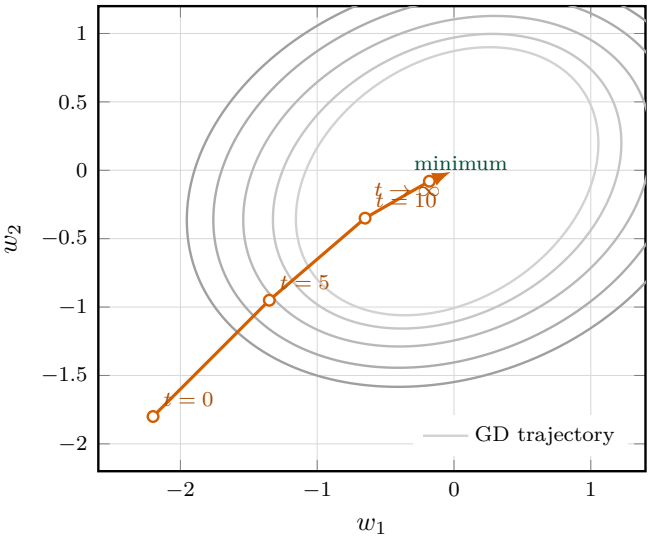

Figure 16: Schematic: Gradient-descent iterates contracting toward the minimizer of a convex quadratic cost. Ellipses are level sets; arrows show the "steepest descent along contours" direction.

Geometry of the logistic surface. The decision rule is linear in feature space even though the posterior itself is smoothly varying. Figure 17 depicts this duality: the white hyperplane slices the space into two half-spaces while the probability "ramp" shows how margins translate into calibrated confidences.

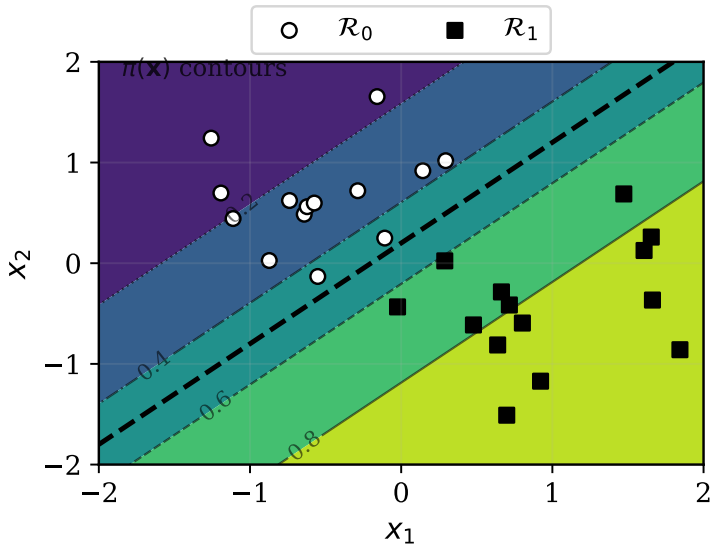

Figure 17: Schematic: Illustrative logistic-regression boundary. The dashed line marks the linear decision boundary at probability 0.5; labeled contours show how the posterior varies smoothly with margin, enabling calibrated decisions and adjustable thresholds.

4.5 Probabilistic Interpretation: MLE and MAP

The ERM view in Chapter 3 treats learning as minimizing an average loss plus (optionally) a regularizer. The probabilistic view arrives at the same objective from a different direction:

- **MLE** maximizes the data likelihood under a chosen observation model (for logistic regression: Bernoulli with $p_i = \sigma(\boldsymbol{\beta}^\top \tilde{\mathbf{x}}_i)$).
- **MAP** maximizes the posterior, which multiplies the likelihood by a prior $p(\boldsymbol{\beta})$. In optimization form, MAP adds a penalty $-\log p(\boldsymbol{\beta})$, which is exactly regularization.

Two common priors explain the two penalties that appear most often in practice: a zero-mean Gaussian prior yields an L2 (ridge) penalty, while a Laplace prior

yields an L1 (lasso) penalty. The schematic below illustrates the MLE→MAP idea on a simple mean-estimation problem: with little data, the prior matters; with enough data, MAP approaches MLE.

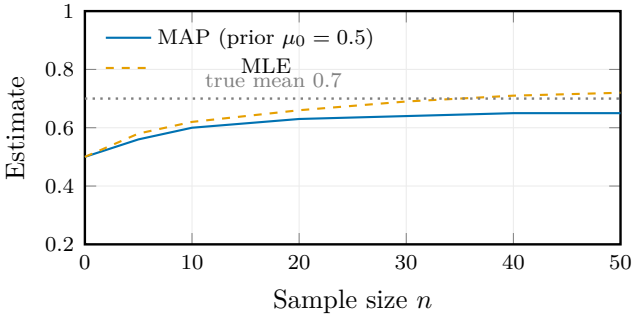

Figure 18: Schematic: MAP estimates interpolate between the prior mean and the data-driven MLE. As the sample size grows, the MAP curve approaches the true mean.

4.6 Confusion Matrices and Derived Metrics

Once we have a probabilistic classifier, we need diagnostics that quantify performance on held-out data. For multi-class prediction, the confusion matrix C_{ij} records the number of examples with true class i predicted as j. From C we compute accuracy, per-class precision/recall, and aggregate metrics. *Macro-averaged* precision/recall first evaluate the metric per class and then average them uniformly, whereas *micro-averaged* precision/recall pool all true/false positives across classes before computing the ratio (equivalent to weighting each example equally). Visual inspection (Figure 20) helps diagnose systematic errors across classes.

On highly imbalanced problems accuracy and AUROC can be misleading; prefer class-balanced metrics (macro-F1) and AUPRC. Figure 19 collects ROC and PR curves on one page so you can choose operating points explicitly.

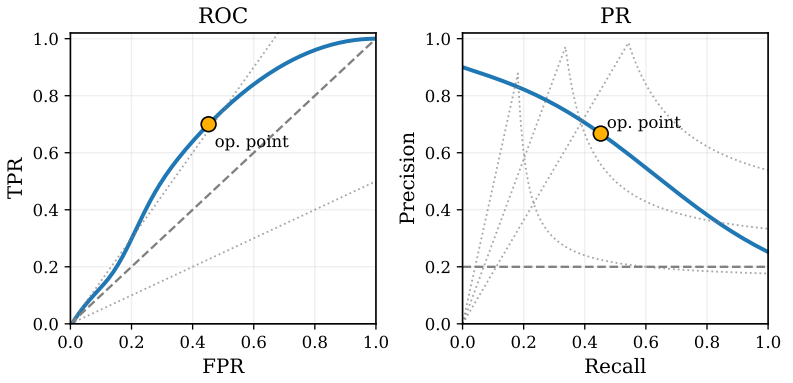

Figure 19: Schematic: ROC and PR curves with an explicit operating point. Left: ROC curve with iso-cost lines; right: PR curve with a class-prevalence baseline and iso-F1 contours. Together they visualize threshold trade-offs and calibration quality.

Imbalance and thresholds

Use class or sample weights (e.g., inverse prevalence) inside the loss, and pick thresholds via ROC/PR curves or explicit cost ratios rather than defaulting to 0.5. With symmetric priors but asymmetric costs, predict class 1 when the logit exceeds $\log(c_{10}/c_{01})$; for rare positives, report PR-AUC alongside AUROC.

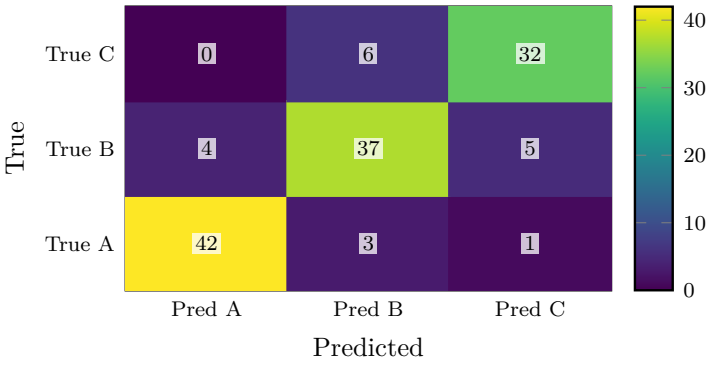

Figure 20: Schematic: Confusion matrix for a three-class classifier; diagonals dominate, indicating strong accuracy with modest confusion between classes B and C.

Key takeaways

- Logistic regression models class probability with a sigmoid link and maximizes a concave log-likelihood (equivalently minimizes a convex negative log-likelihood); there is no closed-form solution.
- ROC and PR curves provide threshold-independent evaluation; AUC summarizes performance.
- Proper feature scaling and regularization improve convergence and generalization.

Probability calibration

Discrimination metrics (ROC/PR, AUC) say how well a classifier ranks examples but not how reliable its probabilities are. Calibration methods such as Platt scaling and temperature scaling adjust the logits so that predicted probabilities match empirical frequencies (e.g., 0.8 scores correspond to $\approx 80\%$ positives), often measured via Expected Calibration Error (ECE) and inspected with reliability diagrams (Platt, 1999; Guo et al., 2017).

Table 3: Schematic: Handling class imbalance for logistic models (Chapter 4 reference table).

Tactic	**When/why**
Stratified splits (and K-fold)	Preserve class ratios in train/validation/test to avoid optimistic validation scores.
Class weighting / cost-sensitive loss	Multiply the cross-entropy (or hinge loss) by per-class weights so minority errors matter more. Useful when collecting more data is difficult.
Resampling (over/undersampling, SMOTE)	Balance the dataset prior to training. Helps tree ensembles and linear models; pair with cross-validation to avoid overfitting. Use simple baselines (logistic/SVM) as a tie-break to detect overfitting.
Threshold tuning	Choose a decision threshold based on PR curves or cost ratios rather than default 0.5; report PR-AUC when positives are rare.

Exercises and lab ideas

- Implement a minimal example from this chapter and visualize intermediate quantities (plots or diagnostics) to match the pseudocode.
- Stress-test a key hyperparameter or design choice discussed here and report the effect on validation performance or stability.
- Re-derive one core equation or update rule by hand and check it numerically against your implementation.

Where we head next. Logistic regression still yields a linear decision boundary. Chapter 5 introduces biologically inspired neuron models and perceptrons as trainable building blocks; stacking nonlinearities breaks the linearity ceiling and sets up multilayer networks and backpropagation.

References. Full citations for works mentioned in this chapter appear in the book-wide bibliography.

Part II: Neural networks, sequence modeling, and NLP

5 Introduction to Neural Networks

Learning Outcomes

After this chapter, you should be able to:

- Describe the core ingredients of neural networks (architecture, activations, learning).
- Explain how multilayer perceptrons learn via gradient-based training.
- Identify common pitfalls (saturation, poor initialization) and basic remedies.

Chapter 4 established linear and logistic models as strong baselines, but their decision boundaries are linear in the original feature space. We now shift from the statistical lens to a biological one: neurons as simple units whose collective behavior yields nonlinear decision boundaries. This chapter introduces neuron models, perceptrons, activation functions, and the first learning rules (perceptron and Adaline). The roadmap in Figure 1 marks this as the start of the neural strand.

Design motif

Biology as engineering abstraction: start with simple units, make the update rule explicit, and use geometry to build intuition before the algebra gets deep.

5.1 Biological Inspiration

Human intelligence is often characterized by behaviors that resemble cognitive processes such as learning, reasoning, and decision-making. To replicate such intelligent behavior artificially, it is natural to look towards the biological systems that exhibit these capabilities. The human brain, composed of billions of interconnected neurons, serves as a primary source of inspiration.

Neurons and Neural Activity A biological neuron can be conceptualized as a processing unit that receives multiple input signals, integrates them, and produces an output signal if certain conditions are met. The key components of a neuron include:

- **Dendrites:** Receive incoming signals from other neurons.
- **Cell body (soma):** Integrates incoming signals.
- **Axon:** Transmits the output signal to other neurons.
- **Synapses:** Junctions where signals are transmitted between neurons.

Signals arriving at the dendrites are often chemical in nature and can excite or inhibit the neuron. When the combined input exceeds a certain threshold, the neuron ”fires,” sending an electrical impulse down the axon to connected neurons. This firing is not simply binary; the strength and timing of signals can influence the neuron’s response.

Complexities and Unknowns Despite extensive research, many aspects of neural function remain poorly understood, including:

- How signals arriving at different dendrites simultaneously interact.
- The effect of signal timing and frequency on neuron firing.
- The mechanisms of cooperation and competition among neurons.
- How neural activity culminates in complex behaviors.

These uncertainties highlight the challenges in directly modeling biological neurons and motivate the development of simplified artificial models.

5.2 From Biological to Artificial Neural Networks

Artificial neural networks (ANNs) are computational models inspired by the structure and function of biological neural systems. The goal is to create systems that can process information, learn from data, and perform tasks that require intelligence.

Key Features of Artificial Neural Networks To design an ANN that captures essential aspects of biological neural processing, several features must be considered:

1. **Architecture:** The arrangement and connectivity of neurons within the network. This includes the number of layers, the pattern of connections (e.g., feedforward, recurrent), and the flow of information.
2. **Signal Propagation:** How input signals are transmitted through the network, transformed by neurons, and produce outputs.
3. **Learning Mechanism:** The method by which the network adjusts its parameters (e.g., weights) based on data to improve performance. This involves capturing and retaining knowledge from experience.
4. **Activation Dynamics:** The rules governing neuron activation, including how neurons decide to fire based on inputs, the degree of activation, and inhibition mechanisms.

Historical Context The concept of artificial neural networks dates back to the early 1940s, with pioneering work that attempted to mathematically model neuron behavior. Over the past eight decades, ANNs have evolved significantly, leading to a variety of architectures and learning algorithms. This evolution reflects ongoing efforts to better approximate biological intelligence and to address practical challenges in computation and learning.

5.3 Outline of Neural Network Study

This chapter introduces perceptrons and common activation functions. Chapter 6 then develops multilayer perceptrons and the backpropagation machinery needed to train them; Chapter 12 returns to sequence models with recurrent connections.

Formal definitions of the perceptron neuron model (activation functions, weighted sums, and thresholds) follow in Section 5.10. Refer back to the biological narrative above when interpreting the abstractions.

5.4 Neural Network Architectures

Neural networks can be broadly categorized based on the flow of information through their structure. Understanding these architectures is crucial for designing and analyzing neural models that mimic biological neural systems.

Feedforward Neural Networks Feedforward neural networks (FNNs) are characterized by a unidirectional flow of information from input to output layers without any cycles or loops. The information propagates forward through successive layers of neurons, each layer transforming the input received from the previous layer.

Conceptually, this can be thought of as a cascade of neuron activations where each neuron receives input signals, processes them, and passes the output to the next layer. This architecture aligns with the idea that sensory information in biological systems is processed in a hierarchical manner.

Mathematically, if we denote the input vector as $\mathbf{x}$, the output of layer l as $\mathbf{a}^{(l)}$, and the weight matrix connecting layer $l-1$ to layer l as $\mathbf{W}^{(l)}$, the

feedforward operation is given by:

$$\mathbf{z}^{(l)} = \mathbf{a}^{(l-1)}\mathbf{W}^{(l)} + \mathbf{b}^{(l)} \tag{5.1}$$

$$\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)}). \tag{5.2}$$

where $\mathbf{b}^{(l)}$ is the bias vector and $f(\cdot)$ is the activation function applied elementwise.

Shapes and convention. We use the row-major (deep-learning) convention. A single example is a row vector $\mathbf{a}^{(l)} \in \mathbb{R}^{1\times n_l}$, a mini-batch stacks examples by rows $A^{(l)} \in \mathbb{R}^{B\times n_l}$, and weights map features by right multiplication $W^{(l)} \in \mathbb{R}^{n_{l-1}\times n_l}$. Biases $b^{(l)} \in \mathbb{R}^{n_l}$ broadcast across the batch: $Z^{(l)} = A^{(l-1)}W^{(l)} + \mathbf{1}(b^{(l)})^\top$. We reserve $\phi(\cdot)$ for kernel feature maps (Appendix B).

Recurrent Neural Networks In contrast, recurrent neural networks (RNNs) allow information to flow in cycles, enabling feedback connections. This means that the network's state at a given time depends not only on the current input but also on previous states, effectively creating a form of memory.

The recurrent architecture is more flexible and biologically plausible since neurons can influence each other bidirectionally and inputs/outputs can be introduced or sampled at various points in the network. This allows modeling of temporal sequences and dynamic behaviors. A simple recurrent update is

$$\mathbf{h}_t = f(\mathbf{x}_t\mathbf{W}_{xh} + \mathbf{h}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h), \qquad \mathbf{y}_t = \mathbf{h}_t\mathbf{W}_{hy} + \mathbf{b}_y,$$

with the full treatment deferred to Chapter 12.

5.5 Activation Functions

Activation functions determine how the input to a neuron is transformed into an output signal, effectively controlling the neuron's excitation level. They play a critical role in enabling neural networks to model complex, nonlinear relationships.

Biological Motivation In biological neurons, excitation occurs when the combined chemical signals exceed a certain threshold, triggering an action potential

(a "fire"). Similarly, artificial neurons use activation functions to decide whether to activate (fire) based on their input.

Common Activation Functions Activation functions map the aggregated input z to a neuron's output $y = f(z)$; they inject nonlinearity and control gradient flow during learning. Different choices trade off biological plausibility, numerical stability, and ease of optimization.

- **Step Function (Heaviside)**:

$$f(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$

  Models a binary firing behavior but is not differentiable, limiting its use in gradient-based learning.

- **Sign Function**:

$$f(x) = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$$

  Allows for inhibitory (negative) outputs, mimicking excitatory and inhibitory neuron behavior. We adopt the convention $f(0) = 0$; some authors either leave sign(0) undefined or set it to $+1$, so it is helpful to state the choice explicitly.

- **Linear Function**:

$$f(x) = x$$

  Useful in some contexts but cannot model nonlinearities alone.

- **Sigmoid Function**:

$$f(x) = \frac{1}{1 + e^{-x}}$$

  Smoothly maps inputs to $(0, 1)$, differentiable, and historically popular. Because sigmoid outputs saturate near 0 and 1, gradients can become small in deep stacks; later chapters discuss practical workarounds and alternatives.

- **Hyperbolic Tangent (tanh)**:

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

  Maps inputs to $(-1, 1)$, zero-centered, often preferred over sigmoid.

- **ReLU (Rectified Linear Unit)**:

$$f(x) = \max(0, x)$$

  Computationally efficient and helps mitigate vanishing gradient problems.

Notation note: activations and thresholds

In this chapter we use $f(\cdot)$ as a generic placeholder for an activation function; when we need the logistic sigmoid specifically we write $\sigma(\cdot)$. Elsewhere in the book, $\phi(\cdot)$ denotes a kernel feature map. For thresholded functions we adopt $H(0) = 1$ (Heaviside) and $\operatorname{sgn}(0) = 0$ by convention. These choices do not affect continuous models but keep examples consistent.

5.6 Learning Paradigms in Neural Networks

When building a neural network, whether feedforward or recurrent, the fundamental process involves producing an output, comparing it with a target, and then adjusting the network parameters based on the error. This iterative process is the essence of *learning*. We distinguish several learning paradigms depending on the availability and nature of the target information:

Supervised Learning In supervised learning, the network is provided with input-output pairs. The network produces an output for a given input, compares it to the known target output, computes an error, and updates its parameters to reduce this error. This requires labeled data and is the most common learning paradigm in practice.

Unsupervised Learning In unsupervised learning, there is no explicit target output. The network must discover patterns or structure in the input data by

itself. This often involves competition among different patterns, where some patterns become dominant and reinforce themselves, while others are suppressed. The network evolves until it reaches an equilibrium state where the learned representations stabilize. Beyond competitive learning, unsupervised methods encompass clustering, density estimation, dimensionality reduction, autoencoders, and modern self-supervised objectives—any setting where structure is inferred directly from the inputs.

Reinforcement Learning Reinforcement learning (RL) models learning from interaction with feedback. An agent with policy $\pi(a \mid s)$ selects actions, collects rewards, and updates π to improve expected return. Full RL treatments appear later; here the point is that not all learning is supervised, and neural-network controllers are common in modern RL.

5.7 Fundamentals of Artificial Neural Networks

The foundational model of artificial neural networks dates back to McCulloch and Pitts (1943), who proposed a simple neuron model capturing essential features of biological neurons.

McCulloch-Pitts Neuron Model Consider a single neuron with multiple binary inputs $x_i \in \{0, 1\}$, $i = 1, \ldots, n$. Each input is associated with a weight w_i, which can be positive (excitatory) or negative (inhibitory). The neuron computes a weighted sum of its inputs:

$$S = \sum_{i=1}^{n} w_i x_i. \tag{5.3}$$

The output y of the neuron is determined by comparing S to a threshold θ:

$$y = \begin{cases} 1, & \text{if } S \geq \theta, \\ 0, & \text{otherwise.} \end{cases} \tag{5.4}$$

Key characteristics of this model include:

- **Binary inputs:** Inputs are either active (1) or inactive (0).
- **Excitatory and inhibitory weights:** Weights $w_i > 0$ excite the neuron, while $w_i < 0$ inhibit it.
- **Thresholding:** The neuron fires (outputs 1) only if the weighted sum exceeds the threshold.

Interpretation This simple neuron can be viewed as a linear classifier that partitions the input space into two regions separated by the hyperplane defined by the equation

$$\sum_{i=1}^{n} w_i x_i = \theta. \tag{5.5}$$

The learning task reduces to finding appropriate weights w_i and threshold θ that correctly classify inputs.

Excitation and Inhibition The neuron can be excited or inhibited depending on the sign and magnitude of the weights. For example:

- If all $w_i > 0$, the neuron is purely excitatory.
- If some $w_i < 0$, those inputs inhibit the neuron.
- The balance of excitation and inhibition determines the neuron's response.

In biological circuits inhibition is carried by specialized interneurons, whereas here a negative weight is an abstract shortcut; the sign simply indicates whether an input pushes the weighted sum above or below the threshold. Artificial neurons are function approximators; similarity to biology is inspirational, not mechanistic.

Learning Objective In this model, learning can be interpreted as adjusting the weights w_i and threshold θ to achieve desired input-output mappings. The challenge is to find these parameters such that the neuron outputs 1 for inputs belonging to a certain class and 0 otherwise.

5.8 Mathematical Formulation of the Neuron Output

To summarize, the neuron output is given by

$$y = f\left(\sum_{i=1}^{n} w_i x_i - \theta\right), \tag{5.6}$$

where $f(\cdot)$ is the activation function, which in the McCulloch-Pitts model is a Heaviside step function:

$$f(z) = \begin{cases} 1, & z \geq 0, \\ 0, & z < 0. \end{cases} \tag{5.7}$$

We explicitly set $f(0) = 1$; other texts sometimes use $f(0) = \frac{1}{2}$, so documenting the convention avoids confusion when comparing derivations. It is also common to absorb the threshold into an augmented weight vector by defining $x_0 = 1$ and $w_0 = -\theta$, yielding a pure inner product $\mathbf{w}^\top \mathbf{x}$ that we will reuse in the perceptron section.

This model laid the groundwork for later developments in neural networks, including the introduction of differentiable nonlinearities that enable gradient-based learning.

5.9 McCulloch-Pitts neuron: examples and limits

Recall that the McCulloch-Pitts (MP) neuron model is defined by a weighted sum of binary inputs compared against a threshold to produce a binary output. Formally, for inputs $x_i \in \{0, 1\}$ and weights w_i, the neuron output y is given by

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i x_i \geq \theta, \\ 0 & \text{otherwise,} \end{cases} \tag{5.8}$$

where θ is the threshold.

Example: AND and OR gates - For an AND gate with inputs x_1, x_2, set weights $w_1 = w_2 = 1$ and threshold $\theta = 2$. The output is 1 only if both inputs

are 1, matching the AND truth table.

- For an OR gate, keep weights $w_1 = w_2 = 1$ but set $\theta = 1$. The output is 1 if at least one input is 1, matching the OR truth table.

This demonstrates how the MP neuron can implement simple logical functions by appropriate choice of weights and threshold.

Limitations of the MP model Despite its conceptual simplicity, the MP neuron has significant limitations:

- **No learning mechanism:** The weights and threshold must be manually assigned or guessed. There is no algorithmic way to adjust parameters based on data.
- **Limited computational power:** The MP neuron can only represent linearly separable functions. Complex patterns requiring nonlinear decision boundaries cannot be modeled.
- **Binary inputs and outputs:** The model is restricted to binary signals, limiting its applicability to real-valued data.

These limitations motivated the development of more sophisticated neuron models and learning algorithms.

5.10 From MP Neuron to Perceptron and Beyond

The MP neuron laid the groundwork for subsequent models that introduced learning capabilities and continuous-valued inputs and outputs.

Perceptron model The perceptron, introduced by Rosenblatt in 1958, extends the MP neuron by incorporating a learning algorithm to adjust weights based on training data. The perceptron output is

$$y = \begin{cases} 1 & \text{if } \mathbf{w}^\top \mathbf{x} + b \geq 0, \\ 0 & \text{otherwise,} \end{cases} \tag{5.9}$$

where $\mathbf{x}$ is the input vector, $\mathbf{w}$ the weight vector, and b the bias (threshold).

Targets and encodings

We switch between labels in `{0,1}` (probability view) and labels in `{-1,+1}` (margin view). Convert with `y_pm = 2*y01 - 1` and `y01 = (y_pm + 1)/2`. Perceptron updates below use the `{-1,+1}` encoding.

The perceptron learning rule iteratively updates weights to reduce classification errors, enabling the model to learn from data rather than relying on manual parameter selection. With labels $y_i \in \{-1, +1\}$, a mistake triggers $\mathbf{w} \leftarrow \mathbf{w} + \eta y_i \mathbf{x}_i$ and $b \leftarrow b + \eta y_i$. The induced separating hyperplane and signed distance are illustrated in Figure 21.

Perceptron update from the signed margin. Let $d_i = y_i(\mathbf{w}^\top \mathbf{x}_i + b)$ be the signed margin. If $d_i \geq 0$ the example is correctly classified; if $d_i < 0$ the example is misclassified. A common perceptron criterion is

$$J(\mathbf{w}, b) = -\sum_{i \in \mathcal{M}} d_i = -\sum_{i \in \mathcal{M}} y_i(\mathbf{w}^\top \mathbf{x}_i + b),$$

where $\mathcal{M}$ is the set of misclassified examples. Taking a gradient step on J yields

$$\mathbf{w} \leftarrow \mathbf{w} + \eta y_i \mathbf{x}_i, \qquad b \leftarrow b + \eta y_i,$$

which is exactly the perceptron update. In augmented form, set $x_0 = 1$ and $w_0 = b$, and the update becomes $\mathbf{w} \leftarrow \mathbf{w} + \eta y_i \mathbf{x}_i$. Geometrically, each mistake nudges the hyperplane so the signed distance d_i increases.

Perceptron convergence theorem. If a training set is linearly separable with margin $\gamma > 0$, the perceptron learning algorithm is guaranteed to find a separating hyperplane after at most $(R/\gamma)^2$ updates, where R bounds the input norms. Rescaling features changes R and γ, so standardizing inputs tightens the bound. When the data are not separable the algorithm can cycle forever; Section 6.1 (and Chapter 6) therefore emphasize feature scaling, bias terms, and the move to differentiable multilayer models to handle nonlinear problems.

Perceptron convergence theorem (proof sketch)

Assume there exists a unit vector $\mathbf{w}^\star$ such that $y_i\,\mathbf{w}^\star \cdot \mathbf{x}_i \geq \gamma$ for all i and that $\|\mathbf{x}_i\| \leq R$. Let w(t) denote the perceptron weights after t mistakes. Each mistake updates:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + y_i \mathbf{x}_i.$$

1. **Progress along the separator.** The inner product with w* grows by at least gamma each mistake:

$$\begin{aligned}\mathbf{w}^{(t+1)} \cdot \mathbf{w}^\star &= \mathbf{w}^{(t)} \cdot \mathbf{w}^\star + y_i\,\mathbf{x}_i \cdot \mathbf{w}^\star \\ &\geq \mathbf{w}^{(t)} \cdot \mathbf{w}^\star + \gamma.\end{aligned}$$

   Thus after T mistakes, the dot product with w* is at least T*gamma.

2. **Bounding the norm.** The squared norm grows slowly:

$$\begin{aligned}\|\mathbf{w}^{(t+1)}\|^2 &= \|\mathbf{w}^{(t)}\|^2 + \|\mathbf{x}_i\|^2 + 2y_i\mathbf{x}_i \cdot \mathbf{w}^{(t)} \\ &\leq \|\mathbf{w}^{(t)}\|^2 + R^2,\end{aligned}$$

   because the mistake condition implies $y_i\mathbf{x}_i \cdot \mathbf{w}^{(t)} \leq 0$. Inductively, $\|\mathbf{w}^{(T)}\|^2 \leq TR^2$.

3. **Combine via Cauchy–Schwarz.**

$$\begin{aligned}T\gamma &\leq \mathbf{w}^{(T)} \cdot \mathbf{w}^\star \\ &\leq \|\mathbf{w}^{(T)}\|\|\mathbf{w}^\star\| \leq \sqrt{T}\,R,\end{aligned}$$

   which implies $T \leq (R/\gamma)^2$.

Therefore the perceptron halts after finitely many mistakes on separable data. If the data are not separable, some $\gamma > 0$ cannot be found, and the above argument no longer applies; hence the need for multilayer networks.

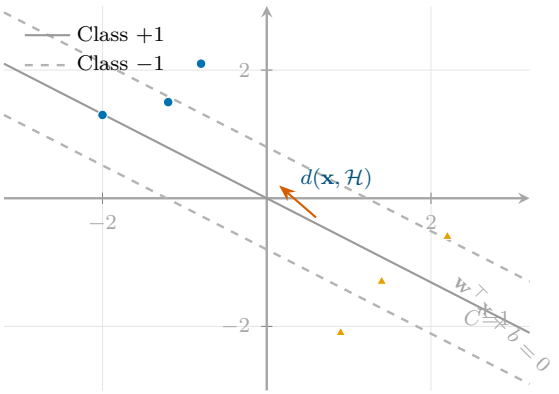

Figure 21: Schematic: Perceptron geometry. Points on either side of the separating hyperplane receive different labels, and the signed distance to the boundary controls both the class prediction and the magnitude of the update during learning. Compare to Figure 17 in Chapter 4: both share a linear separator, but logistic smooths the boundary into calibrated probabilities.

Common perceptron pitfalls

- **Feature scaling:** Large-magnitude features dominate updates; standardize inputs first.
- **Random seed sensitivity:** Different initial weights can lead to drastically different separating hyperplanes.
- **Non-separable data:** Without slack variables or kernels the perceptron will not converge; diagnose this before training indefinitely. XOR is the canonical counterexample.

Adaline model The Adaptive Linear Neuron (Adaline), developed in the 1960s, further improves on the perceptron by using a linear activation function and minimizing a continuous error function (mean squared error). This allows the use of gradient descent for training, leading to more stable convergence.

Adaline weight update (derivation) Adaline uses a linear output $y = \mathbf{w}^\top \mathbf{x} + b$ and the squared error

$$E = \tfrac{1}{2}(t - y)^2.$$

The gradient is $\partial E/\partial \mathbf{w} = -(t-y)\mathbf{x}$ and $\partial E/\partial b = -(t-y)$, so the update is

$$\mathbf{w} \leftarrow \mathbf{w} + \eta(t-y)\mathbf{x}, \qquad b \leftarrow b + \eta(t-y).$$

Unlike the perceptron, Adaline updates on every example and scales the step by the residual $t-y$; this is the first explicit appearance of gradient-based weight optimization in the neural narrative.

The perceptron and Adaline models are limited to linearly separable problems. To overcome this, multilayer perceptrons (MLPs) with hidden layers were introduced; Chapter 6 and Chapter 7 develop the mechanics in full.

Perceptron vs. logistic regression

Linear score $s(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$. The perceptron predicts $\mathbb{1}[s \geq 0]$ and updates $\mathbf{w} \leftarrow \mathbf{w} + \eta y_i \mathbf{x}_i$ (and $b \leftarrow b + \eta y_i$) only on mistakes, with $y_i \in \{-1, +1\}$. Logistic regression predicts $\sigma(s)$, minimizes cross-entropy $-\sum_i y_i \log \sigma(s_i) + (1-y_i)\log(1-\sigma(s_i))$, and steps by $\sum_i (\sigma(s_i) - y_i)\mathbf{x}_i$. Prefer logistic when calibrated probabilities and smooth optimization are needed (Chapter 4).

Author's note: what a single perceptron cannot express

A single perceptron makes one global, all-or-none decision: one hyperplane, one threshold, one set of weights shared across every example. That simplicity is the point of the model, but it also explains its limitations. Many real problems require *communities* of units that specialize: different hidden units respond to different regions, features, or patterns, and their combined vote produces a flexible decision surface. Multi-layer networks do not just add parameters; they add internal structure that lets different parts of the model "care about" different parts of the data.

Key takeaways

- The perceptron and Adaline turn threshold units into trainable classifiers by updating weights from data.
- Geometry (hyperplanes and signed distance) explains predictions and update magnitude.
- Logistic regression keeps the same linear score but learns calibrated probabilities via a smooth loss (Chapter 4).
- Nonlinear tasks (e.g., XOR) require multilayer networks and backpropagation (Chapters 6 to 7).

Exercises and lab ideas

- Implement a minimal example from this chapter and visualize intermediate quantities (plots or diagnostics) to match the pseudocode.
- Stress-test a key hyperparameter or design choice discussed here and report the effect on validation performance or stability.
- Re-derive one core equation or update rule by hand and check it numerically against your implementation.

Where we head next. Perceptrons are intentionally simple: hard thresholds and uniform updates. Seeing both what they can do (linear separation) and what they cannot do (nonlinear tasks such as XOR) clarifies why multilayer networks are needed. Chapter 6 builds directly on the perceptron/Adaline story by chaining units into the smallest multi-layer network, defining a performance (loss) function, and then asking the key question: *how should weights change to improve performance?* That question forces the chain rule and motivates smooth activations. Chapter 7 then generalizes the same bookkeeping to arbitrary depth and turns the derivation into an efficient training algorithm.

References. Full citations for works mentioned in this chapter appear in the book-wide bibliography.

6 Multi-Layer Perceptrons: Challenges and Foundations

Learning Outcomes

After this chapter, you should be able to:

- Build the smallest multi-layer network and write its forward equations.
- Define a simple performance (loss) function for the network output.
- Explain why gradient descent is the right tool for weight updates.
- Show why hard thresholds block gradients and motivate smooth activations.
- Derive the weight updates for the two-neuron network using the chain rule.

Chapter 5 introduced the perceptron and Adaline: single units that learn by updating weights from data, with Adaline giving our first explicit glimpse of gradient descent on a smooth performance function. In this chapter we keep the story linear and concrete. We build the smallest possible network (two neurons in series), define a performance function, and ask the core question: *how should the weights change to improve performance?* Answering that question forces us to use derivatives (the chain rule) and leads naturally to gradient descent. Along the way we encounter a practical obstacle: hard thresholds are not differentiable, so they do not support gradient-based learning. We replace them with smooth activations and immediately gain a clean update story. This is the conceptual bridge to full backpropagation in Chapter 7. The roadmap in Figure 1 shows this as the hinge between single-unit models and multilayer training.

Design motif

Build the smallest trainable network you can, keep every intermediate quantity visible, and let the chain rule explain how learning signals flow.

A short roadmap: building a trainable network

The chapter follows one tight loop:

- **Build:** write the forward computation (a two-neuron chain).
- **Judge:** define a performance function P (we use squared error).
- **Move:** use derivatives to update parameters via gradient descent.
- **Fix:** choose a differentiable activation so those derivatives exist and carry signal.

Once you can execute this loop for two neurons, scaling to many neurons is mostly bookkeeping (Chapter 7).

How this chapter fits the workflow

The same objective→audit workflow from the supervised toolkit still applies; what changes is the *representation*.

- From Chapter 3: diagnostics (learning curves, bias–variance) tell you *what* is going wrong.
- From Chapter 4: a linear probabilistic baseline tells you *how far* you can go without nonlinear features.
- Here: we build the smallest nonlinear network and derive its gradient updates, setting up the general backprop machinery in Chapter 7.

6.1 From a single unit to the smallest network

Function estimation as the unifying view. Learning is function estimation: approximate some unknown mapping $f : X \to Y$ from examples. A neural network is a structured way to represent a nonlinear f by composing simple units. In this chapter we keep the bookkeeping minimal and use one tiny network plus a simple squared error objective so we can focus on the mechanics of learning.

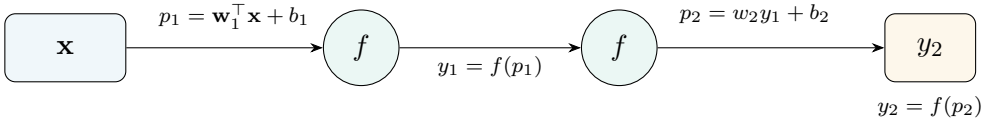

Figure 22: Schematic: The minimal neural network used in this chapter is a two-neuron chain. The first unit produces an intermediate signal, and the second unit maps that signal to the final output.

From one unit to a chain of units. A perceptron computes a weighted sum and then applies an activation (often a threshold in the classical presentation):

$$y = f(p), \qquad p = \mathbf{w}^\top \mathbf{x} + b, \tag{6.1}$$

where $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{w} \in \mathbb{R}^n$, and b is a bias (threshold). Because the boundary $\mathbf{w}^\top \mathbf{x} + b = 0$ is a hyperplane, a single unit can only represent linear separations. This explains the classic XOR failure and motivates building a network of units.

The smallest network that is more than a single unit is a *two-neuron chain*: one neuron feeds another. Write

$$p_1 = \mathbf{w}_1^\top \mathbf{x} + b_1, \qquad y_1 = f(p_1), \tag{6.2}$$

$$p_2 = w_2 y_1 + b_2, \qquad y_2 = f(p_2). \tag{6.3}$$

Even this tiny network introduces the central idea of neural networks: intermediate computations (here y_1) are reused and influence the final output y_2. A single neuron is a linear classifier; chaining neurons gives a nonlinear representation. With multiple hidden units, that added structure is enough to solve XOR.

Author's note: why a network changes the story

With one perceptron, every training example pushes on the same single separator. A network introduces *intermediate representations*: different hidden units can respond to different patterns, and the output unit can combine those responses. That added structure is what lets neural networks model nonlinearity while still using simple building blocks.

A checklist of what we must settle (and why)

To turn "a diagram of neurons" into something trainable, we need:

- **A parameterization:** weights and biases that control the mapping from inputs to outputs.
- **A performance function:** a scalar P that is lower when the output is better.
- **An update rule:** a systematic way to change parameters to reduce P.
- **Differentiability:** if we want to use gradient descent, every link from parameters to P must be differentiable so the chain rule can propagate credit (and blame).

The chapter keeps everything small so you can see all four ingredients in one place.

Bias as a learned threshold. A hard threshold θ can be absorbed into a bias term by writing $p = \mathbf{w}^\top \mathbf{x} - \theta$ and setting $b = -\theta$. In practice we append $x_0 = 1$ and treat b as another weight w_0; the algebra is identical. The bias handles *where* the unit switches, while the weights handle *which direction* it prefers.

6.2 Performance: what are we trying to improve?

Once we have a forward computation, we need a performance function that tells us whether the output is good. For one training example with target t, a simple choice is the squared error

$$P = \frac{1}{2}(y_2 - t)^2. \tag{6.4}$$

The factor $\frac{1}{2}$ makes derivatives cleaner. If you prefer to *maximize* a score rather than minimize an error, you could take $-\frac{1}{2}(y_2 - t)^2$ instead. The math below is identical up to a sign. We will minimize P.

Why a square? The signed error $e = y_2 - t$ can be positive or negative. Squaring removes the sign and penalizes large deviations more heavily, while keeping P smooth so a small change in a weight produces a small change in

performance. That smoothness is exactly what makes derivative-based updates meaningful.

Author's note: one objective is enough for the first derivation

We only need a performance function that (1) is easy to differentiate and (2) rewards outputs that move toward the target. The squared error does both, so it is a good stand-in while we learn the mechanics. Once the chain rule story is clear, swapping in other objectives is mostly a matter of changing a few local derivatives at the output layer.

A geometric intuition. For a fixed input, the performance becomes a surface over the weights. If the surface looks like a "bowl," then the bottom is the optimum. The goal is to move weights along this surface in the direction that improves performance.

6.3 Gradient descent: how do weights move?

We now ask: how should $\mathbf{w}_1, w_2, b_1, b_2$ change to reduce P? The standard answer is gradient descent:

$$\theta \leftarrow \theta - \eta \nabla_\theta P, \tag{6.5}$$

where θ stands for any parameter and $\eta > 0$ is the step size. Geometrically, you can picture the performance surface as a landscape: the gradient points uphill, so we step in the opposite direction to descend toward a minimum. The step size controls how far we move; too large can overshoot, too small can crawl.

For a weight vector, the update is a vector step:

$$\Delta \mathbf{w} = -\eta \nabla_{\mathbf{w}} P. \tag{6.6}$$

This is the "move the weights in the right direction" story made precise: we do not guess the direction; we compute it from the derivative of performance. Importantly, we update *all* weights at once (a vector step), not one coordinate at a time.

Step size is a design choice. The gradient gives a direction; the step size η sets the distance. In practice you pick η small enough to avoid oscillation and large enough to make progress. Chapter 7 returns to this choice (learning-rate

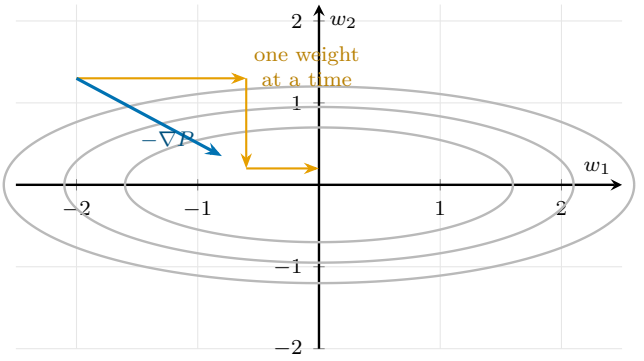

Figure 23: Schematic: Think of performance as a surface over the weights. Gradient descent moves in one vector step (blue), whereas coordinate-wise updates can zig-zag (orange).

schedules, momentum, and other practical stabilizers) once the core derivative story is solid.

6.4 Why hard thresholds block learning

At this point the story is simple: define P, compute ∇P, and update weights. The catch is that computing ∇P requires derivatives through the activation. When we apply the chain rule to Equation (6.3), factors like $f'(p_1)$ and $f'(p_2)$ appear immediately.

If f is a hard threshold (a step function), it is discontinuous and non-differentiable at the threshold. That breaks the gradient story: $f'(p)$ either does not exist or is zero almost everywhere, so derivatives cannot guide learning. This is the core reason we replace thresholds with *smooth, differentiable activations*.

Absorbing the threshold. The threshold itself can be folded into a bias term, but the discontinuity remains. We remove the discontinuity by choosing a smooth f.

6.5 Differentiable activations and the sigmoid trick

A classic choice is the logistic (sigmoid) function

$$\sigma(p) = \frac{1}{1 + e^{-p}}. \tag{6.7}$$

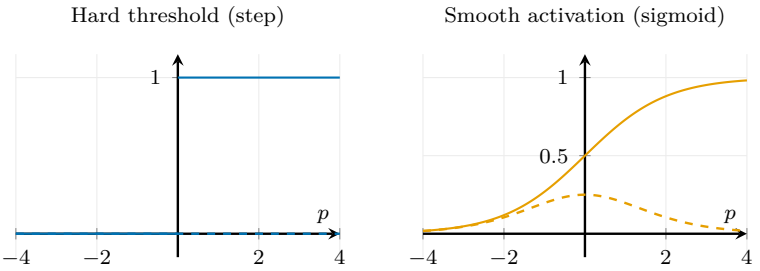

Figure 24: Schematic: Hard thresholds block gradient-based learning because the derivative is zero almost everywhere. A smooth activation like the sigmoid provides informative derivatives across a wide range of inputs.

It maps real inputs to $(0, 1)$ and is differentiable everywhere. The key identity is

$$\sigma'(p) = \sigma(p)\,[1 - \sigma(p)]. \tag{6.8}$$

This is a useful trick: the derivative is a function of the *output* itself. If $y = \sigma(p)$ is already computed in the forward pass, then $\sigma'(p) = y(1 - y)$ is immediately available in the backward pass. No extra exponentials are needed.

Author's note: the derivative is already in the forward pass

In practice you rarely want to recompute expensive expressions during learning. For the sigmoid, once you have computed the output `y = sigmoid(p)`, you also have its slope for free: `sigmoid_prime = y*(1-y)`. This is a small example of a bigger pattern: backpropagation works because we cache intermediate results on the forward pass and reuse them on the backward pass.

Derivation sketch. Let $\beta = \sigma(\alpha) = (1 + e^{-\alpha})^{-1}$. Differentiate:

$$\frac{d\beta}{d\alpha} = \frac{e^{-\alpha}}{(1 + e^{-\alpha})^2} = \left(\frac{1}{1 + e^{-\alpha}}\right)\left(1 - \frac{1}{1 + e^{-\alpha}}\right) = \beta(1 - \beta).$$

This is the exact algebraic shortcut used in neural networks.

6.6 Deriving weight updates for the two- neuron network

The diagram in Figure 22 is also a derivative map: to update a weight, follow how a small change in that weight would flow forward to the output and then back to the performance. The chain rule turns that story into algebra.

We now compute the gradients in Equation (6.3) using the chain rule. First note the easy derivatives:

- $\dfrac{\partial P}{\partial y_2} = y_2 - t.$
- $\dfrac{\partial y_i}{\partial p_i} = f'(p_i).$
- $\dfrac{\partial p_2}{\partial w_2} = y_1$ and $\dfrac{\partial p_2}{\partial y_1} = w_2.$
- $\dfrac{\partial p_1}{\partial \mathbf{w}_1} = \mathbf{x}.$

Second layer.

$$\frac{\partial P}{\partial w_2} = \frac{\partial P}{\partial y_2}\frac{\partial y_2}{\partial p_2}\frac{\partial p_2}{\partial w_2} = (y_2 - t)\, f'(p_2)\, y_1, \tag{6.9}$$

and similarly

$$\frac{\partial P}{\partial b_2} = (y_2 - t) f'(p_2). \tag{6.10}$$

First layer. The first layer feels the effect of the second layer through the chain rule:

$$\frac{\partial P}{\partial \mathbf{w}_1} = \frac{\partial P}{\partial y_2}\frac{\partial y_2}{\partial p_2}\frac{\partial p_2}{\partial y_1}\frac{\partial y_1}{\partial p_1}\frac{\partial p_1}{\partial \mathbf{w}_1} \tag{6.11}$$

$$= (y_2 - t)\, f'(p_2)\, w_2\, f'(p_1)\, \mathbf{x}, \tag{6.12}$$

with bias derivative

$$\frac{\partial P}{\partial b_1} = (y_2 - t) f'(p_2) w_2 f'(p_1). \tag{6.13}$$

Error terms (backprop view). Define

$$\delta_2 := \frac{\partial P}{\partial p_2} = (y_2 - t) f'(p_2). \tag{6.14}$$

Then $\partial P/\partial w_2 = \delta_2 y_1$ and $\partial P/\partial b_2 = \delta_2$. The first layer receives a backpropagated error

$$\delta_1 := \frac{\partial P}{\partial p_1} = \delta_2 w_2 f'(p_1), \tag{6.15}$$

so $\partial P/\partial \mathbf{w}_1 = \delta_1 \mathbf{x}$ and $\partial P/\partial b_1 = \delta_1$.

This is the central lesson: once we compute a local error term, it can be reused across many gradients. That reuse is exactly what makes backpropagation efficient and is why deeper networks remain tractable.

Worked example: one numerical gradient step (sanity check)

Take a single input $x = [1, -1]^\top$, target $t = 1$, sigmoid activation $f = \sigma$, and parameters $\mathbf{w}_1 = [0.8,\ 0.2]^\top$, $b_1 = 0$, $w_2 = 1$, $b_2 = 0$.

Forward: $p_1 = 0.6$, $y_1 = \sigma(p_1) \approx 0.646$; $p_2 = y_1$, $y_2 = \sigma(p_2) \approx 0.656$; $P = \frac{1}{2}(y_2 - t)^2 \approx 0.059$.

Backward: $\sigma'(p) = y(1-y)$, so $\delta_2 = (y_2 - t)\sigma'(p_2) \approx -0.078$ and $\delta_1 = \delta_2 w_2 \sigma'(p_1) \approx -0.018$. Thus $\nabla_{\mathbf{w}_1} P = \delta_1 x \approx [-0.018,\ +0.018]^\top$ and $\nabla_{w_2} P = \delta_2 y_1 \approx -0.050$.

Update: with $\eta = 0.5$, gradient descent increases w_2 slightly (since the gradient is negative) and nudges $\mathbf{w}_1$ in a direction that increases y_2 toward the target.

6.7 From two neurons to multi- layer networks

Nothing essential changes for deeper networks; we simply apply the same chain rule repeatedly. For a layer l with pre-activations $\mathbf{p}^{(l)}$ and weights $W^{(l+1)}$, the error signal satisfies

$$\delta^{(l)} = \left(\delta^{(l+1)} (W^{(l+1)})^\top\right) \circ f'(\mathbf{p}^{(l)}), \tag{6.16}$$

where $\circ$ denotes element-wise multiplication. This recursion is the heart of backpropagation, which we derive and operationalize in Chapter 7.

Author's note: what backprop adds

Conceptually, nothing new happens when you go from two neurons to many layers: it is still the chain rule and the same local derivatives. What changes is the *organization*: we run a forward pass that caches intermediate values, then a backward pass that reuses those caches to compute all gradients efficiently (and stably) for an entire batch. Chapter 7 turns the recursion into an implementable algorithm and shows the standard bookkeeping.

6.8 Summary

- A two-neuron chain is the smallest network that goes beyond a single perceptron.
- Learning starts by defining a performance function and asking how weights change it.
- Gradient descent uses derivatives to choose the correct update direction.
- Hard thresholds obstruct gradients; smooth activations fix the problem.
- The sigmoid derivative $\sigma'(p) = \sigma(p)(1 - \sigma(p))$ is a convenient identity because it reuses the output.
- The two-neuron derivation already contains the backpropagation pattern used in deep networks.

Key takeaways

- Training is a loop: define a forward computation, define a scalar performance, then use derivatives to update parameters.
- Hard thresholds break the gradient story; smooth activations (e.g., sigmoid) restore informative derivatives.
- The two-neuron derivation already contains the reusable "local error" pattern that scales to deep networks.

Exercises and lab ideas

Setup. These reinforce the two-neuron derivation and prepare you for the multi-layer bookkeeping in Chapter 7.

- **Numerical gradient check:** Implement finite differences for the two-neuron chain and compare to your analytic gradients; report relative error.
- **Step vs. sigmoid:** Replace the smooth activation with a hard threshold and observe what breaks when you try to compute updates via derivatives.
- **XOR with two hidden units:** Train a tiny MLP on XOR and plot its decision regions; note sensitivity to initialization and step size.

Where we head next. Chapter 7 generalizes this derivation to arbitrary depth and shows how to implement the backward pass efficiently for batches.

References. Full citations for works mentioned in this chapter appear in the book-wide bibliography.

7 Backpropagation Learning in Multi-Layer Perceptrons

Learning Outcomes

- Derive the layerwise backpropagation recursions for arbitrary-depth MLPs.
- Connect theoretical gradients to implementation details (vectorization, caching, numerical stability).
- Translate training diagnostics (learning curves, early stopping) into concrete optimization policies.

Building on the two-neuron derivation in Chapter 6, we now derive backpropagation as a systematic application of the chain rule to an L-layer network. The key idea is unchanged: compute local error terms (the δ's), then reuse them

to obtain all weight gradients efficiently. The roadmap in Figure 1 marks this chapter as the training engine for deep models.

Design motif

Compute local error signals once, then reuse them to update every parameter efficiently. The organization (cache forward values, then sweep backward) is the algorithm.

Author's note: gradients move vectors, not single weights

A practical MLP rarely updates one coordinate at a time; gradients are treated as full vectors so that every weight moves coherently. The backpropagation formulas derived below exist precisely to deliver those vector updates efficiently.

Why backprop matters

Backpropagation is what turns a multilayer network from a diagram into a trainable model. It reuses local error signals so you can compute *all* gradients efficiently, which makes it practical to learn hidden representations (not just tune a final linear layer), and to pair that increased capacity with the same validation→audit discipline (learning curves, early stopping, and slice checks).

7.1 Context and Motivation

Recall that a single-layer perceptron has no hidden layer (just input and output), while a shallow network adds one hidden layer between them. These shallow models solve linearly separable problems but remain insufficient for more complex tasks. The multi-layer perceptron introduces multiple layers of neurons, each connected by weighted links, enabling the network to learn nonlinear decision boundaries.

However, this increased complexity raises the question: *How do we update the weights across multiple layers to minimize the error at the output?* Unlike the single-layer perceptron, where weight updates depend directly on the output error, in a deep network, changes in weights at one layer propagate through subsequent layers, influencing the final output in a nonlinear and intertwined manner.

7.2 Problem Setup

Consider a multi-layer perceptron with layers indexed by $l = 1, 2, \ldots, L$, where L is the output layer. Each layer l contains neurons indexed by i, and the output of neuron i in layer l is denoted by $a_i^{(l)}$. The input to this neuron before activation is denoted by $z_i^{(l)}$. The weights connecting neuron i in layer $l-1$ to neuron j in layer l are denoted by $w_{ij}^{(l)}$.

The forward pass through the network is given by:

$$z_j^{(l)} = \sum_i a_i^{(l-1)} w_{ij}^{(l)} + b_j^{(l)}, \tag{7.1}$$

$$a_j^{(l)} = f\big(z_j^{(l)}\big), \tag{7.2}$$

where $b_j^{(l)}$ is the bias term for neuron j in layer l, and $f(\cdot)$ is the activation function, typically nonlinear (e.g., sigmoid, ReLU). Equation (7.1) makes it explicit that we sum over every incoming neuron i in layer $l-1$ to form the affine pre-activation $z_j^{(l)}$.

7.3 Loss and Objective

To keep the story linear (and aligned with Chapter 6), we will use a simple squared-error objective. Let the network output be $\mathbf{a}^{(L)}$ and let $\mathbf{t}$ be the target (one-hot targets for classification are fine; we do not need a separate regression/-classification split yet). A standard loss is

$$\mathcal{L} = \frac{1}{2} \sum_k \left(t_k - a_k^{(L)} \right)^2 . \tag{7.3}$$

The goal of learning is to adjust the weights $\{w_{ij}^{(l)}\}$ to minimize $\mathcal{L}$. Later in this chapter, we briefly note how common alternatives (notably cross-entropy with sigmoid/softmax outputs) simplify the output-layer error term; the backprop recursion itself does not change.

7.4 Challenges in Weight Updates

In a shallow network, weight updates can be computed directly from the output error. However, in a deep network, the output error depends on all weights in a complex way. A change in a weight in an earlier layer affects the activations of

subsequent layers, ultimately influencing the output.

For example, consider a weight $w_{ij}^{(l)}$ connecting neuron i in layer $l-1$ to neuron j in layer l. Changing this weight affects $z_j^{(l)}$, which affects $a_j^{(l)}$, which in turn affects all neurons in layers $l+1, l+2, \ldots, L$. Therefore, the total effect of changing $w_{ij}^{(l)}$ on the loss is a composition of many intermediate effects.

7.5 Notation for Layers and Neurons

To formalize this, we introduce the following notation:

- l: layer index, with $l = 0$ representing the input layer, and $l = L$ the output layer.
- i: neuron index in layer $l-1$.
- j: neuron index in layer l.
- k: neuron index in layer L (output layer).
- $a_i^{(l)}$: activation of neuron i in layer l.
- $z_j^{(l)}$: weighted input to neuron j in layer l.
- $w_{ij}^{(l)}$: weight from neuron i in layer $l-1$ to neuron j in layer l.
- $b_j^{(l)}$: bias of neuron j in layer l.
- $f(\cdot)$: activation function.

These definitions carry directly into the forward-pass recap below, where we chain the affine map and nonlinearity across layers.

7.6 Forward Pass Recap

The forward pass computes activations layer by layer:

$$z_j^{(l)} = \sum_i a_i^{(l-1)} w_{ij}^{(l)} + b_j^{(l)}, \tag{7.4}$$

$$a_j^{(l)} = f\big(z_j^{(l)}\big). \tag{7.5}$$

The output layer activations $a_k^{(L)}$ are compared to the

Mini example: two-layer backprop in practice

```
# Shapes: X in R^{B x d}, W1 in R^{d x h}, W2 in R^{h x c}
def step(X, Y, params, eta, wd=1e-4, p_drop=0.1):
    W1, b1, W2, b2 = params
    B = X.shape[0]
    # Forward pass
    Z1 = X @ W1 + b1
    H1 = relu(Z1)
    mask1 = (np.random.rand(*H1.shape) > p_drop).astype(
        H1.dtype
    )
    # inverted dropout
    H1 = H1 * mask1 / (1 - p_drop)
    Z2 = H1 @ W2 + b2
    Yhat = softmax(Z2)
    # Backward pass
    delta2 = (Yhat - Y) / B            # CE output error
    grad_W2 = H1.T @ delta2
    grad_W2 += wd * W2              # L2 decay
    grad_b2 = delta2.sum(axis=0)
    delta1 = (delta2 @ W2.T) * relu_deriv(Z1)
    # dropout backprop
    delta1 = delta1 * mask1 / (1 - p_drop)
    grad_W1 = X.T @ delta1
    grad_W1 += wd * W1
    grad_b1 = delta1.sum(axis=0)
    # SGD step
    return (W1 - eta * grad_W1, b1 - eta * grad_b1,
            W2 - eta * grad_W2, b2 - eta * grad_b2)
```

The elementwise product $*$ mirrors the Hadamard notation from Equations (7.1) to (7.2). This miniature example bridges the algebra to vectorized code before we scale to L-layer MLPs later in the chapter.

Shape ledger for an L-layer MLP (batch size B)

- $A^{(l-1)} \in \mathbb{R}^{B \times n_{l-1}},\ Z^{(l)}, \delta^{(l)} \in \mathbb{R}^{B \times n_l}$
- $W^{(l)} \in \mathbb{R}^{n_{l-1} \times n_l},\ b^{(l)} \in \mathbb{R}^{n_l}$
- $\partial L / \partial W^{(l)} = (A^{(l-1)})^\top \delta^{(l)} / B \in \mathbb{R}^{n_{l-1} \times n_l}$
- $\partial L / \partial b^{(l)} = \text{batch\_mean}(\delta^{(l)}) \in \mathbb{R}^{n_l}$

Layers share this structure; convolutional/sequence models reuse the same calculus with different bookkeeping.

7.7 Backpropagation: Recursive Computation of Error Terms

Recall that our goal is to compute the gradient of the loss with respect to the weights in the network, specifically for weights connecting layer l to layer $l+1$. We denote the weight connecting neuron i in layer l to neuron j in layer $l+1$ as $w_{ij}^{(l)}$.

We will continue with the squared-error loss from Chapter 6:

$$\mathcal{L} = \frac{1}{2} \sum_k (t_k - a_k^{(L)})^2. \tag{7.6}$$

where t_k is the target output and $a_k^{(L)}$ is the activation of output neuron k. Other losses change only a few local derivatives (most notably at the output layer), but the backprop recursion and bookkeeping are the same.

To update the weights using gradient descent, we need to compute

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}}.$$

Chain rule decomposition By the chain rule, we have

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_j^{(l+1)}} \cdot \frac{\partial z_j^{(l+1)}}{\partial w_{ij}^{(l)}}. \tag{7.7}$$

where $z_j^{(l+1)}$ is the weighted input to neuron j in layer $l+1$:

$$z_j^{(l+1)} = \sum_i a_i^{(l)} w_{ij}^{(l)} + b_j^{(l+1)}.$$

Here $a_i^{(l)}$ is the activation of neuron i in layer l, and $b_j^{(l+1)}$ the bias term.

Since $z_j^{(l+1)}$ is linear in $w_{ij}^{(l)}$, we have

$$\frac{\partial z_j^{(l+1)}}{\partial w_{ij}^{(l)}} = a_i^{(l)}.$$

Thus,

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \delta_j^{(l+1)} a_i^{(l)}, \tag{7.8}$$

where we define the *error term*

$$\delta_j^{(l+1)} := \frac{\partial \mathcal{L}}{\partial z_j^{(l+1)}}.$$

Collecting the $\delta_j^{(l+1)}$ for all neurons in layer $l+1$ forms a vector $\boldsymbol{\delta}^{(l+1)}$ with the same dimension as $z^{(l+1)}$, ensuring the gradient $\frac{\partial \mathcal{L}}{\partial W^{(l)}}$ has the same shape as the weight matrix.

Interpretation of $\delta_j^{(l+1)}$ The term $\delta_j^{(l+1)}$ measures how sensitive the error is to changes in the net input $a_j^{(l+1)}$. Our task reduces to computing these δ terms for all neurons in the network.

7.7.1 Output layer error terms

For the output layer L, the activation of neuron k is

$$a_k^{(L)} = f\big(z_k^{(L)}\big),$$

where $f(\cdot)$ is the activation function.

The error term for output neuron k is

$$\delta_k^{(L)} = \frac{\partial \mathcal{L}}{\partial z_k^{(L)}} \tag{7.9}$$

$$= \frac{\partial \mathcal{L}}{\partial a_k^{(L)}} \frac{\partial a_k^{(L)}}{\partial z_k^{(L)}} \tag{7.10}$$

$$= \big(a_k^{(L)} - t_k\big)\, \phi'\big(z_k^{(L)}\big), \tag{7.11}$$

where ϕ' denotes the derivative of the activation function evaluated element-wise. For cross-entropy with sigmoid/softmax output, ϕ' cancels and $\delta_k^{(L)} = a_k^{(L)} - t_k$; for MSE retain the factor above.

7.7.2 Hidden layer error terms

For a hidden neuron j in layer l, the error term $\delta_j^{(l)}$ depends on the error terms of the neurons in the next layer $l+1$ to which it connects. Using the chain rule,

$$\delta_j^{(l)} = \frac{\partial \mathcal{L}}{\partial z_j^{(l)}} \tag{7.12}$$

$$= \sum_k \frac{\partial \mathcal{L}}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \tag{7.13}$$

$$= \sum_k \delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}. \tag{7.14}$$

Since

$$z_k^{(l+1)} = \sum_m a_m^{(l)}\, w_{mk}^{(l)} + b_k^{(l+1)},$$

and $a_j^{(l)} = \phi\big(z_j^{(l)}\big)$, we have

$$\frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = w_{jk}^{(l)}\, \phi'\big(z_j^{(l)}\big).$$

Substituting into (7.14) yields

$$\delta_j^{(l)} = \phi'\big(z_j^{(l)}\big) \sum_k w_{jk}^{(l)} \delta_k^{(l+1)}. \tag{7.15}$$

For sigmoid activations ϕ, the derivative simplifies to $\phi'(z_j^{(l)}) = a_j^{(l)}(1 - a_j^{(l)})$; other activations require substituting their respective derivatives in (7.15).

Summary: Backpropagation recursion Backpropagation is reverse-mode automatic differentiation on the network graph. A forward pass caches intermediates; a reverse pass reuses caches to get all gradients in $O(P)$ time (versus $O(P^2)$ for finite differences) for P parameters. Frameworks (PyTorch/JAX/TF) automate this; the algebra below is its manual derivation.

7.8 Backpropagation Algorithm: Detailed Derivation

Recall that the goal of backpropagation is to compute the gradient of the loss with respect to each weight in the network, enabling gradient descent updates. Consider a single neuron k in the output layer with output o_k and activation a_k. The target output is t_k.

Error function and its derivatives We use the squared-error loss for a single output neuron:

$$\mathcal{L}_{\text{SE}} = 0.5\,(t_k - o_k)^2. \tag{7.16}$$

Our objective is to compute $\frac{\partial \mathcal{L}_{\text{SE}}}{\partial w_{jk}}$, where w_{jk} is the weight from neuron j in the previous layer to neuron k.

By the chain rule,

$$\frac{\partial \mathcal{L}_{\text{SE}}}{\partial w_{jk}} = \frac{\partial \mathcal{L}_{\text{SE}}}{\partial o_k} \cdot \frac{\partial o_k}{\partial a_k} \cdot \frac{\partial a_k}{\partial w_{jk}}. \tag{7.17}$$

Step 1: Derivative of error with respect to output From the squared-error definition above,

$$\frac{\partial \mathcal{L}_{\text{SE}}}{\partial o_k} = o_k - t_k. \tag{7.18}$$

Step 2: Derivative of output with respect to activation Assuming the activation function f is the sigmoid,

$$o_k = f(a_k) = \frac{1}{1 + e^{-a_k}},$$

its derivative is

$$\frac{\partial o_k}{\partial a_k} = f'(a_k) = o_k(1 - o_k). \tag{7.19}$$

Step 3: Derivative of activation with respect to weight The activation a_k is the weighted sum of inputs:

$$a_k = \sum_j w_{jk} x_j.$$

Here x_j is the output from neuron j in the previous layer. Thus,

$$\frac{\partial a_k}{\partial w_{jk}} = x_j. \tag{7.20}$$

Putting it all together Substituting (7.18), (7.19), and (7.20) into (7.17):

$$\frac{\partial \mathcal{L}_{\text{SE}}}{\partial w_{jk}} = (o_k - t_k)\, o_k(1 - o_k)\, x_j. \tag{7.21}$$

Define the *error signal* for neuron k as

$$\delta_k = (o_k - t_k)\, o_k(1 - o_k). \tag{7.22}$$

Then,

$$\frac{\partial \mathcal{L}_{\text{SE}}}{\partial w_{jk}} = \delta_k x_j. \tag{7.23}$$

The gradient descent update therefore becomes

$$\Delta w_{jk} = -\eta\, \delta_k x_j, \tag{7.24}$$

where η is the learning rate.

7.9 Backpropagation for Hidden Layers

For neurons in hidden layers, the error signal δ_j is computed by propagating the error backward from the next layer. Consider a hidden neuron j with pre-activation z_j and activation $o_j = f(z_j)$. Its error signal is

$$\delta_j = o_j(1 - o_j) \sum_k w_{jk} \delta_k, \tag{7.25}$$

where the sum is over all neurons k in the next layer to which neuron j connects.

The weight update for weights w_{ij} feeding into neuron j is then

$$\Delta w_{ij} = -\eta\, \delta_j x_i, \tag{7.26}$$

where x_i is the output from the previous layer neuron i.

7.10 Batch and Stochastic Gradient Descent

Given a training set of N examples $\{(x^{(n)}, t^{(n)})\}_{n=1}^{N}$, the weight updates can be computed in different ways:

- **Batch gradient descent:** Compute the gradient over the entire dataset and update weights once per epoch:

$$\Delta w = -\frac{\eta}{N} \sum_{n=1}^{N} \delta^{(n)} x^{(n)}.$$

- **Stochastic gradient descent (SGD):** Update weights after each training example using the instantaneous gradient $-\eta\, \delta^{(n)} x^{(n)}$. Although the updates are noisy, SGD often converges faster in practice and can escape shallow local minima.

Optimizer and stability notes

SGD remains the backbone; momentum and Adam/AdamW from Chapter 11 accelerate convergence. Add L2 weight decay to the gradient or decouple it (AdamW) to avoid biasing adaptive steps. For deep or ill-conditioned nets, gradient clipping can prevent explosions; for classification, pair these with cross-entropy and the log-sum-exp stability tricks introduced alongside CNNs in Chapter 11. Reverse-mode AD underlies all of these updates.

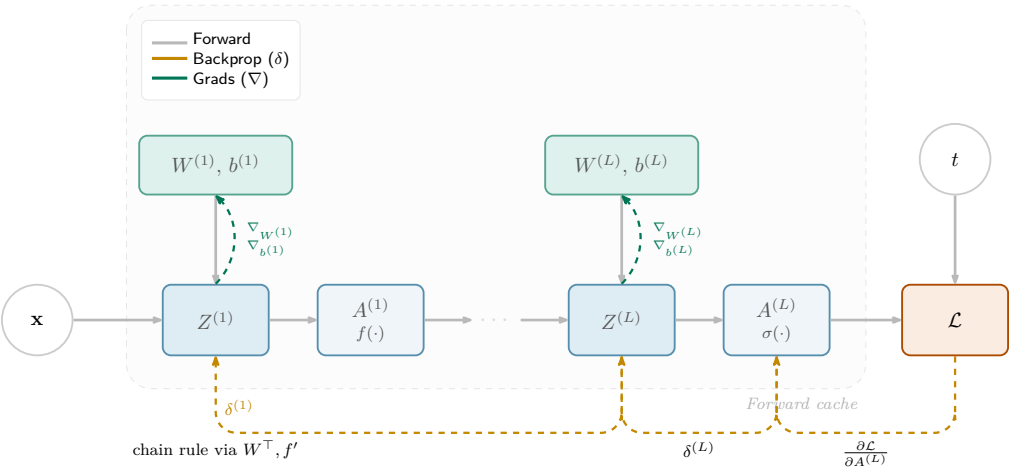

Figure 25: Schematic: Computational graph for a feedforward network. Backpropagation is reverse-mode AD: the forward sweep caches intermediate values; the reverse sweep propagates error signals (deltas) and accumulates gradients for weights and biases from those cached values.

Debugging and gradient-check checklist

- **Overfit a tiny batch:** Ensure the loss can be driven near zero on a handful of samples.
- **Gradient norms:** Track $\|\nabla W^{(l)}\|$ per layer; look for dead layers (all zeros) or explosions.
- **Finite-difference check:** Compare analytic gradients to numerical finite differences on a tiny network with fixed seeds; relative error should be $< 10^{-6}$.
- **Shape assertions:** Verify that $Z^{(l)}, A^{(l)}, \delta^{(l)}$ have the expected batch shapes and that bias broadcasts correctly.
- **Layerwise sanity:** For a one-layer linear model, backprop gradients should match the closed-form linear-regression gradients.

7.11 Backpropagation Algorithm: Brief Numerical Check

For a quick sanity check, take a tiny 2–2–1 network with sigmoid output and cross-entropy loss. Using

$$W^{(1)} = \begin{bmatrix} 0.5 & -0.3 \\ 0.8 & 0.2 \end{bmatrix}, \quad b^{(1)} = [0.1, -0.2],$$
$$W^{(2)} = \begin{bmatrix} 0.7 \\ -0.4 \end{bmatrix}, \quad b^{(2)} = 0.05,$$
$$\mathbf{x} = [0.6, -1.2], \quad t = 1,$$

the forward pass yields

$$z^{(1)} = [-0.56, -0.62], \quad a^{(1)} = [0.3635, 0.3498], \quad z^{(2)} = 0.1646, \quad a^{(2)} = 0.5411,$$

with loss $\mathcal{L} \approx 0.6142$. The cross-entropy output error is $\delta^{(2)} = a^{(2)} - t = -0.4590$. Backpropagating gives

$$\delta^{(1)} = [-0.0743, 0.0418], \quad \nabla_{W^{(2)}} = [-0.1669, -0.1605]^\top, \quad \nabla_{b^{(2)}} = -0.4590,$$

and

$$\nabla_{W^{(1)}} = \begin{bmatrix} -0.0446 & 0.0251 \\ 0.0892 & -0.0501 \end{bmatrix}, \quad \nabla_{b^{(1)}} = [-0.0743, 0.0418].$$

Finite-difference checks on the same network match to numerical precision, validating the implementation.

Aside: squared-error loss (alternative) The remainder of this subsection sketches the classic squared-error backprop derivation as a separate reminder; it is *not* a continuation of the cross-entropy numerical check above.

The error at the output neuron is:

$$e = y - t, \tag{7.27}$$

and the squared error is:

$$\mathcal{L}_{\text{SE}} = \frac{1}{2} e^2. \tag{7.28}$$

Backward Propagation of Error Define the error term δ_j for each neuron j as:

$$\delta_j = e_j \sigma'(net_j), \tag{7.29}$$

where e_j is the error at neuron j, and

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)).$$

is the derivative of the sigmoid function.

For the output neuron:

$$\delta_{\text{out}} = (y_{\text{out}} - t)\, y_{\text{out}}(1 - y_{\text{out}}).$$

For hidden neurons, the error term is computed by backpropagating the weighted sum of the downstream error terms:

$$\delta_j = y_j(1 - y_j) \sum_k w_{kj}\, \delta_k, \tag{7.30}$$

where the sum is over neurons k in the next layer and w_{kj} denotes the weight from neuron j to neuron k.

Weight Update Rule Weights are updated using gradient descent with momentum:

$$\Delta w_{ij}(n) = -\eta\, \delta_j x_i + \gamma \Delta w_{ij}(n-1), \tag{7.31}$$

where

- η is the learning rate,
- γ is the momentum coefficient (typically $0 \leq \gamma < 1$),
- $\Delta w_{ij}(n-1)$ is the previous weight change,
- n indexes the update step (e.g., the current training example in stochastic gradient descent).

The leading negative sign ensures that the update follows the negative gradient direction because each δ_j equals $\partial \mathcal{L}_{\text{SE}} / \partial z_j$.

The new weight is then:

$$w_{ij}(n) = w_{ij}(n-1) + \Delta w_{ij}(n).$$

Interpretation of Learning Rate and Momentum

- The **learning rate** η controls the step size in the weight update. A small η leads to slow convergence, while a large η can cause oscillations or divergence.
- The **momentum** term γ helps smooth the updates by incorporating a fraction of the previous weight change, reducing oscillations and potentially accelerating convergence.

Step-by-Step Example

1. **Initialization:** Draw weights $w_{ij} \sim \mathcal{N}(0, \sigma^2)$ with σ set by He/Xavier rules; set biases to zero and $\Delta w_{ij}(0) = 0$.
2. **Feedforward:** Compute net_j and y_j for all neurons.
3. **Compute output error:** Calculate δ_{out}.
4. **Backpropagate error:** Compute δ_j for hidden neurons.
5. **Update weights:** Use equation (7.31) to update all weights.
6. **Repeat:** Iterate over all training patterns until error E is below threshold or maximum epochs reached.

Mini-batch backprop with explicit regularization

Inputs: mini-batch $\{\mathbf{x}_b, \mathbf{t}_b\}_{b=1}^{B}$, learning rate η, L2 coefficient λ, dropout keep probability $q = 1 - p$.

1. **Forward pass:** propagate activations layer by layer; for each hidden layer draw a dropout mask $\mathbf{m} \sim \text{Bernoulli}(q)$, apply $\tilde{\mathbf{a}} = \mathbf{m} \odot \mathbf{a}/q$, and cache $\mathbf{m}$ for the backward step.
2. **Backward pass:** compute $\nabla_{W^{(\ell)}} \mathcal{L}$ using the cached activations/masks so that dropped units contribute zero gradient.
3. **Update block (per layer):**

$$\mathbf{g}_\ell = \frac{1}{B} \nabla_{W^{(\ell)}} \mathcal{L} + \lambda W^{(\ell)}, \qquad W^{(\ell)} \leftarrow W^{(\ell)} - \eta\, \mathbf{g}_\ell.$$

Biases skip the weight-decay term. With Adam/SGD+momentum, $\mathbf{g}_\ell$ replaces the raw gradient inside the optimizer step so L2 regularization and dropout are always enforced explicitly.

Remarks

- Monitor the training error over epochs; a plateau may indicate the need to adjust learning rate or introduce regularization.
- Shuffle training patterns between epochs when using SGD to avoid cyclic behaviors.
- Always track validation error to detect overfitting and decide when to stop training.

7.12 Training Procedure and Epochs in Multi-Layer Perceptrons

Recall that during training of a multi-layer perceptron (MLP), we iteratively update the weights based on each training pattern. The process for one epoch can be summarized as follows:

1. Present the first input pattern to the network.
2. Perform a forward pass to compute the output.

3. Calculate the error between the actual output and the desired output.
4. Use backpropagation to compute the gradients and update the weights accordingly.
5. Repeat steps 1–4 for all training patterns.

After completing one epoch (i.e., one full pass through all training patterns), we evaluate the overall error. If the error is greater than a predefined tolerance, we continue training for additional epochs until the error converges below the threshold or a maximum number of epochs is reached.

Remarks:

- The weight updates after each pattern are typically small adjustments aimed at reducing the error.
- The initial weights strongly influence the convergence behavior and final solution.
- This iterative process is computationally intensive but essential for learning complex mappings.

7.13 Role and Design of Hidden Layers

In an MLP, the architecture consists of an input layer, one or more hidden layers, and an output layer. The hidden layers are crucial because they enable the network to learn nonlinear mappings.

Key Questions Regarding Hidden Layers:

- **How many hidden layers should be used?** There is no fixed rule; it depends on the complexity of the problem.
- **How many neurons per hidden layer?** This choice affects the network's capacity and generalization ability.
- **What activation functions to use in each layer?** Different layers can use different activation functions, such as sigmoid, ReLU, or tanh.

Design Considerations:

- **Number of neurons:** More neurons increase the capacity to learn complex functions but also increase the risk of overfitting and computational cost.
- **Number of layers:** Deeper networks can represent more complex functions but are harder to train.
- **Activation functions:** Choice affects gradient flow and convergence.

Ultimately, these design choices are made by the practitioner based on experimentation, domain knowledge, and validation performance.

Trade-offs:

- **Too many neurons/layers:** Requires more training data to avoid overfitting; increases computational burden.
- **Too few neurons/layers:** Limits the network's ability to approximate complex functions.

7.14 Case Study: Learning the Function $y = x \sin x$

Consider the problem of training an MLP to approximate the function

$$y = x \sin x.$$

Setup:

- Generate a dataset of input-output pairs $\{(x_i, y_i)\}$ where $y_i = x_i \sin x_i$.
- Use this dataset to train an MLP regressor.
- Evaluate the network's ability to generalize by testing on inputs not seen during training.

Questions to Explore:

- How many hidden layers and neurons per layer are needed to approximate this nonlinear function well?
- What activation functions yield better performance?

- How does the size of the training set affect generalization?

Remarks:

- This is a regression problem, not a classification problem.
- The function is nonlinear and periodic, which challenges the network's approximation capabilities.
- Experimentation with different architectures and hyperparameters is essential.

As a representative example (illustrative; depends on initialization and the training recipe), a two-hidden-layer MLP with widths $[64, 32]$, ReLU activations, Adam optimization, and early stopping on a validation split can achieve mean absolute error on the order of 10^{-3} on held-out samples when trained on 2,000 uniformly spaced points in $[-3\pi, 3\pi]$.

7.15 Applications of Multi-Layer Perceptrons

Multi-layer perceptrons have found widespread applications across various domains due to their ability to approximate complex nonlinear functions. Some notable applications include:

- **Signal processing:** Noise reduction, filtering, and feature extraction.
- **Weather forecasting:** Modeling complex atmospheric patterns.
- **Data compression:** Dimensionality reduction and encoding.
- **Pattern recognition:** Handwriting recognition, face detection.
- **Financial market prediction:** Time series forecasting and anomaly detection.
- **Image recognition:** Object detection and classification.
- **Voice recognition:** Speech-to-text and speaker identification.

Summary: MLPs are versatile and powerful tools that serve as foundational building blocks in many machine learning systems.

7.16 Limitations of Multi-Layer Perceptrons

Despite their versatility, MLPs have several limitations that practitioners must be aware of:

- **Convergence to local minima:** Due to the non-convex nature of the loss surface, training may converge to different local minima depending on the initial weights.
- **Sensitivity to initialization:** Different random initializations can lead to significantly different outcomes.
- **Hyperparameter tuning:** Learning rates, momentum, and regularization require careful tuning for stable convergence.

7.17 Conclusion of Multi-Layer Perceptron Derivations

In this final segment of the chapter, we complete the derivations and discussions related to the multi-layer perceptron (MLP) and its learning algorithm, backpropagation.

Recall that the MLP consists of multiple layers of neurons, each performing an affine transformation followed by a nonlinear activation. The key to training the MLP is to minimize a loss $\mathcal{L}$ defined over outputs and targets.

Backpropagation Algorithm Recap The backpropagation algorithm efficiently computes the gradient of the loss function with respect to all network parameters by applying the chain rule of calculus through the network layers. For a network with L layers, denote by:

$$Z^{(l)} = A^{(l-1)}W^{(l)} + \mathbf{1}(b^{(l)})^\top, \quad A^{(l)} = \phi^{(l)}(Z^{(l)}),$$

where $W^{(l)}$ and $b^{(l)}$ are the weights and biases of layer l, $A^{(l-1)}$ is the previous layer activation (rows are samples), and $\phi^{(l)}$ is the activation function.

The error term at layer l is defined as:

$$\delta^{(l)} = \frac{\partial \mathcal{L}}{\partial Z^{(l)}}.$$

Using the chain rule, the error terms propagate backward as:

$$\delta^{(L)} = \nabla_{A^{(L)}} \mathcal{L} \odot \phi^{(L)\prime}(Z^{(L)}), \tag{7.32}$$

$$\delta^{(l-1)} = \left(\delta^{(l)}(W^{(l)})^\top\right) \odot \phi^{(l-1)\prime}(Z^{(l-1)}), \quad l = L, \ldots, 2, \tag{7.33}$$

where $\odot$ denotes element-wise multiplication and $\phi^{(l)\prime}$ is the derivative of the activation function at layer l.

The gradients of the loss with respect to the parameters are then:

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = (A^{(l-1)})^\top \delta^{(l)}, \tag{7.34}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(l)}} = \mathbf{1}^\top \delta^{(l)}. \tag{7.35}$$

These gradients are used in gradient-based optimization methods (e.g., stochastic gradient descent) to update the parameters and minimize the loss. Figure 26 complements the algebra by showing how cached activations (blue) line up with the backward error signals (orange) in a simple two-layer network.

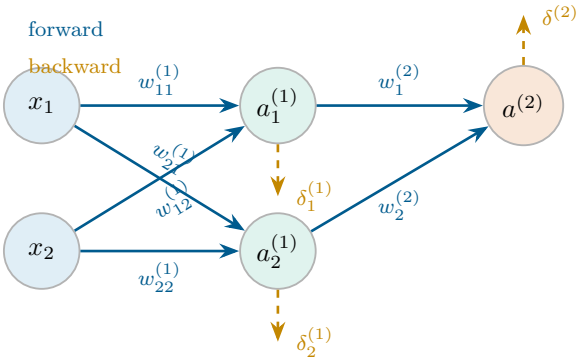

Figure 26: Schematic: Forward (blue) and backward (orange) flows for a two-layer MLP. The cached activations and the layerwise error terms (deltas) are exactly the quantities carried along these arrows; backward signals are computed with the next-layer weights and the activation derivative.

Example Execution An example was provided illustrating the forward pass computation of activations and the backward pass calculation of gradients for a simple MLP with one hidden layer. This example concretely demonstrated how

the chain rule is applied layer-by-layer and how the error signals are propagated backward.

Remarks on Convergence and Practical Considerations While the backpropagation algorithm provides the exact gradients for the MLP, practical training involves additional considerations such as:

- Initialization of weights to avoid vanishing or exploding gradients.
- Choice of activation functions (e.g., ReLU, sigmoid, tanh) affecting gradient flow.
- Regularization techniques (dropout, weight decay) to prevent overfitting.
- Optimization algorithms (momentum, Adam) to accelerate convergence.

These topics will be explored in subsequent chapters; for now, we compare the canonical activation choices in one place.

Comparing canonical nonlinearities With the full MLP and backpropagation machinery in place, it is useful to compare the most common nonlinearities side-by-side. Figure 27 overlays the step, sigmoid, tanh, and ReLU curves so the saturation regions and derivative behavior are visually apparent before we move on to deeper architectures.

For reference, $\sigma'(z) = \sigma(z)\big(1-\sigma(z)\big)$, $\tanh'(z) = 1-\tanh^2(z)$, and the ReLU derivative is 0 for negative inputs and 1 for positive inputs (take 0 at the origin).

Trade-offs While some activation functions are inspired by biological neurons, others are chosen for mathematical convenience and training efficiency. Sigmoid and tanh saturate at large magnitude inputs, which slows gradients in deep networks. ReLU avoids saturation on the positive side but can produce "dying ReLUs" when biases push units negative and the gradients become zero; if many units stall, use He initialization, reduce the learning rate, or swap to a leaky ReLU with a small negative slope (e.g., 0.01). This closes the core backpropagation story for MLPs. Next we summarize practical stability considerations and the key takeaways that guide real training.

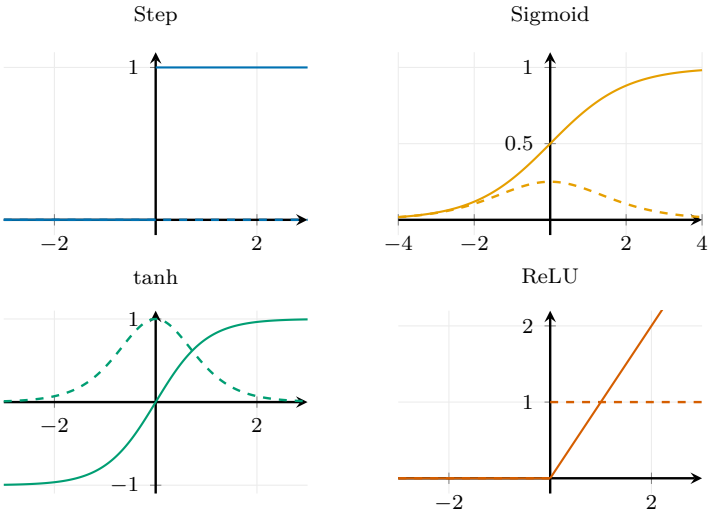

Figure 27: Schematic: Canonical activation functions on a common axis. Solid curves show the activation; dashed curves show its derivative.

Key takeaways

- MLP training relies on stable optimization: proper initialization, learning-rate schedules, and normalization help.
- Regularization (weight decay, dropout) reduces overfitting; validation curves guide early stopping.
- Despite power, MLPs face local minima and sensitivity to hyperparameters.

Practical early stopping and checkpointing

- Maintain a validation split distinct from the training mini-batches. After each epoch, record the validation loss $L_{\text{val}}^{(e)}$.
- Stop training when L_{val} has not improved for k consecutive epochs (typical patience $k \in [5, 10]$). Optionally require a minimum relative improvement (e.g., 0.1%) to smooth noise.
- Always checkpoint the parameters that achieved the best validation score and restore them before testing; averaging the last m checkpoints ("Polyak averaging") can further stabilize performance.

Exercises and lab ideas

- Implement a minimal example from this chapter and visualize intermediate quantities (plots or diagnostics) to match the pseudocode.
- Stress-test a key hyperparameter or design choice discussed here and report the effect on validation performance or stability.
- Re-derive one core equation or update rule by hand and check it numerically against your implementation.

Where we head next. Chapter 8 introduces radial basis function networks, an alternative nonlinear model where the hidden layer is a (mostly) fixed basis expansion and the output layer is trained by a linear solve. It is a useful contrast to end-to-end backpropagation.

References. Full citations for works mentioned in this chapter appear in the book-wide bibliography.

8 Radial Basis Function Networks (RBFNs)

Learning Outcomes

- Explain the architecture and training stages of RBF networks (center selection, width tuning, linear solve).
- Relate RBF solutions to linear estimators (normal equations, pseudoinverse, Wiener filtering) and know when ridge regularization is needed.
- Compare RBFNs to kernelized methods and other nonlinear classifiers to choose appropriate models in practice.

Building on the multilayer perceptron (MLP) architecture (Chapter 6) and its training machinery (Chapter 7), we now introduce a different class of feedforward models: *Radial Basis Function Networks* (RBFNs). The roadmap in Figure 1 places this as the kernel/prototype branch alongside the MLP path.

Design motif

Make the nonlinearity explicit: use a fixed (or lightly tuned) basis expansion in the hidden layer, then learn the output weights with linear-algebra tools.

How to read this chapter

- **Core thread (RBFNs):** architecture → basis functions and widths → linear solve (least squares / ridge) → kernel view and practical tuning.
- **Optional bridge:** a short Wiener-filter refresher connects the RBFN linear solve to classical linear estimation; it is provided for context and can be skimmed without loss of continuity.

8.1 Overview and Motivation

An RBFN is a special category of feedforward neural network characterized by the following properties:

- It has exactly three layers: an input layer, a single hidden layer, and an output layer.

- The input layer simply forwards the raw feature vector to every hidden unit; there are no trainable weights on these connections because the hidden units encode their own parameters (centers and widths).
- The hidden layer applies a nonlinear transformation to the input vector via a set of radial basis functions.
- The output layer is a linear combination of the hidden layer outputs, with trainable weights.

This architecture contrasts with MLPs, which can have multiple hidden layers and trainable weights on all connections.

The RBFN was originally developed as a method to model nonlinear static processes by mapping data from a lower-dimensional input space to a higher-dimensional feature space. The key idea is that data which are *not linearly separable* in the original input space can become *linearly separable* after a suitable nonlinear transformation into a higher-dimensional space. This concept is closely related to the kernel trick used in support vector machines (SVMs); Appendix B collects the classical kernel/SVM viewpoint in one place.

Chapter 3 frames this as a bias–variance tuning problem (choose capacity, regularization, and diagnostics via learning curves). Kernel methods such as kernel ridge regression and SVMs interpret the same trade-off through an RBF kernel matrix; here we keep the bases explicit, then connect to the dual/kernel view later in the chapter.

Author's note: centers come from clustering

The hidden layer of an RBF network is easiest to understand when its centers are viewed as K-means prototypes: pick a coverage of the input space that reflects the data distribution, assign widths accordingly, and let the output layer learn the linear weights on top of those prototypes. Unsupervised clustering up front makes the later supervised solve far more stable.

8.2 Architecture of RBFNs

The RBFN consists of three layers:

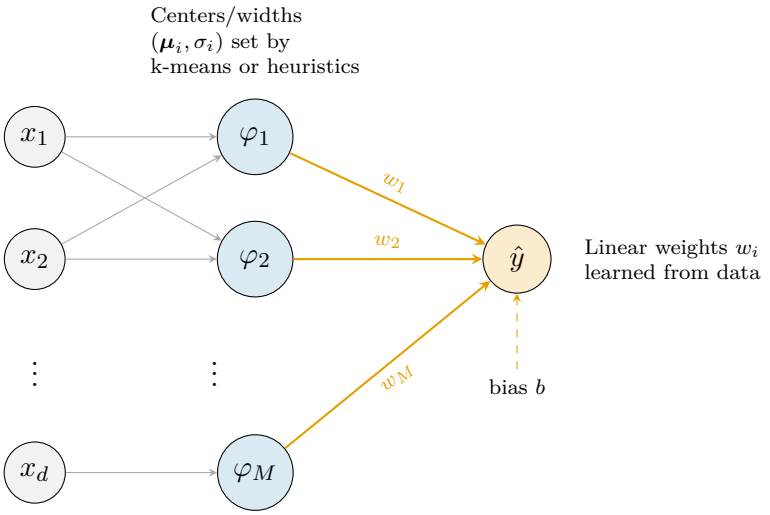

Figure 28: Schematic: RBFN architecture. Inputs feed fixed radial units parameterized by centers and widths; a linear readout with weights (and bias) is trained by a regression or classification loss. Only the output weights are typically learned; centers/widths come from k-means or spacing heuristics.

Notation and shapes

We denote each basis response by $\varphi_i(\mathbf{x})$; stacking them yields $\mathbf{G}(\mathbf{x}) \in \mathbb{R}^M$ with entries $G_i(\mathbf{x}) = \varphi_i(\mathbf{x})$. For a dataset of N samples, the corresponding design matrix $\Phi \in \mathbb{R}^{N \times M}$ stacks one transformed sample per row, with entries $\Phi_{ji} = \varphi_i(\mathbf{x}_j) = G_i(\mathbf{x}_j)$. This matches the design-matrix convention used in Chapter 3.

A picture to keep in mind Once you have the architecture in mind, it helps to visualize what the hidden layer *does*. In one dimension, you can literally draw the bases as overlapping Gaussian bumps; the model output is a weighted sum of those bumps. Figure 29 is the mental model we will reuse as we introduce centers, widths, and the final linear solve.

1. **Input layer:** Receives the input vector $\mathbf{x} \in \mathbb{R}^n$.
2. **Hidden layer:** Applies a set of M nonlinear radial basis functions $\{G_i(\mathbf{x})\}_{i=1}^M$ to the input. These functions serve as feature mappings.

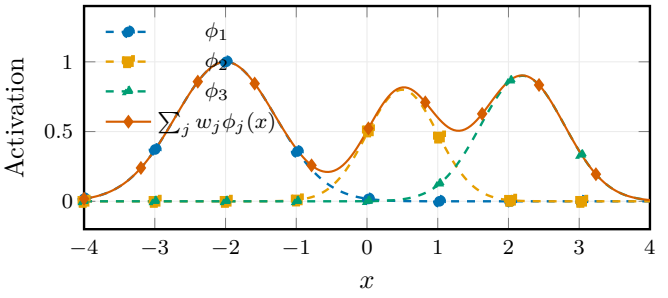

Figure 29: Schematic: Localized Gaussian basis functions (dashed) and their weighted sum (solid). Overlapping bumps allow RBF networks to interpolate complex signals smoothly.

3. **Output layer:** Computes a weighted sum of the hidden layer outputs to produce the final output vector $\mathbf{y} \in \mathbb{R}^K$.

The key distinction is that the input-to-hidden layer connections do not have trainable weights; instead, the hidden layer units themselves perform nonlinear transformations of the input.

8.2.1 Mathematical Formulation

Let the input vector be $\mathbf{x} \in \mathbb{R}^n$. The hidden layer computes the vector

$$\mathbf{G}(\mathbf{x}) = \begin{bmatrix} G_1(\mathbf{x}) \\ G_2(\mathbf{x}) \\ \vdots \\ G_M(\mathbf{x}) \end{bmatrix} \in \mathbb{R}^M.$$

where each $G_i(\mathbf{x})$ is a radial basis function centered at some point $\mathbf{c}_i \in \mathbb{R}^n$; stacking all M responses into $\mathbf{G}(\mathbf{x})$ makes it clear that M controls the dimensionality of the transformed feature space.

The output layer then computes

$$\mathbf{y}(\mathbf{x}) = \mathbf{W}^\top \mathbf{G}(\mathbf{x}) + \mathbf{b}, \tag{8.1}$$

where $\mathbf{W} \in \mathbb{R}^{M \times K}$ is the weight matrix connecting the hidden layer to the output layer, and $\mathbf{b} \in \mathbb{R}^K$ is a bias vector.

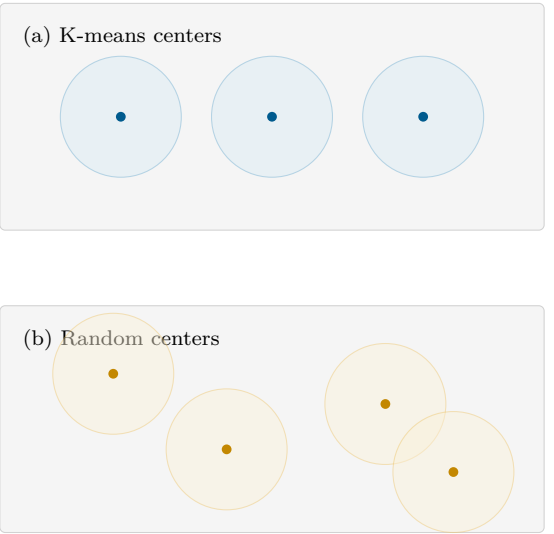

Figure 30: Schematic: Center placement and overlap. Top: K-means prototypes roughly tile the data manifold, giving even overlap; bottom: random centers can leave gaps or excessive overlap, influencing the width (sigma) choice and conditioning.

Interpretation: The hidden layer maps the input $\mathbf{x}$ into a new feature space via nonlinear functions G_i, and the output layer performs a linear combination of these features to produce the final output.

8.3 Radial Basis Functions

The functions $G_i(\mathbf{x})$ are typically chosen to be radially symmetric functions centered at $\mathbf{c}_i$, such as Gaussian functions:

$$G_i(\mathbf{x}) = \varphi\left(\|\mathbf{x} - \mathbf{c}_i\|\right) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{c}_i\|^2}{2\sigma_i^2}\right), \tag{8.2}$$

where σ_i is the width (spread) parameter controlling the receptive field of the i-th basis function.

Other choices of radial basis functions are possible, but the Gaussian is the most common due to its smoothness and locality properties.

Normalized RBFs. Some texts normalize the hidden responses as $\tilde{G}_i(\mathbf{x}) = G_i(\mathbf{x}) / \sum_j G_j(\mathbf{x})$ to smooth predictions when center density is uneven; the linear

readout then uses $\tilde{\mathbf{G}}(\mathbf{x})$ in place of $\mathbf{G}(\mathbf{x})$.

8.4 Key Properties and Advantages

- **Nonlinear transformation without weights:** The input-to-hidden layer mapping is fixed by the choice of centers $\{\mathbf{c}_i\}$ and widths $\{\sigma_i\}$, not by trainable weights.
- **Linear output layer:** Training reduces to finding the optimal weights $\mathbf{W}$ in a linear model, which can be done efficiently using linear regression techniques.
- **Universal approximation:** With sufficiently many radial basis functions placed densely over a compact domain (and with nondegenerate widths), RBFNs can approximate any continuous function to arbitrary accuracy (Park and Sandberg, 1991; Micchelli, 1986).
- **Interpretability:** Each hidden unit corresponds to a localized region in input space, making it easier to understand which prototypes influence a given prediction.

Curse of dimensionality. In high dimensions Euclidean distances concentrate, so widths and center counts must scale with dimension; kernel ridge regression or learned features (e.g., CNNs) often dominate for images/audio.

8.5 Transforming Nonlinearly Separable Data into Linearly Separable Space

Recall from the previous discussion that certain datasets are not linearly separable in their original input space. However, by applying nonlinear transformations, we can map the data into a new feature space where linear separation becomes possible.

Consider a nonlinear transformation function $g(\cdot)$ applied to the input vector $\mathbf{x} \in \mathbb{R}^n$, producing a transformed vector $\mathbf{g}(\mathbf{x}) \in \mathbb{R}^m$. The goal is to find a weight vector $\mathbf{w} \in \mathbb{R}^m$ such that the linear combination $\mathbf{w}^\top \mathbf{g}(\mathbf{x})$ separates the classes.

Example Setup: - Input vectors: $\mathbf{x} \in \{0,1\}^2$ (e.g., $(0,0), (0,1), (1,0), (1,1)$) - Two neurons in the hidden layer, each associated with weight vectors $\mathbf{v}_1$ and $\mathbf{v}_2$.

- Activation functions $g_1(\mathbf{x})$ and $g_2(\mathbf{x})$ correspond to these neurons. - Output is a linear combination of these activations:

$$y = \mathbf{w}^\top \mathbf{g}(\mathbf{x}) = w_1 g_1(\mathbf{x}) + w_2 g_2(\mathbf{x}).$$

Assumptions: - For simplicity, set $\sigma^2 = 1$ (so $2\sigma^2 = 2$) in the Gaussian kernel activation function. - Assume $\mathbf{v}_1 = (0,0)^\top$ and $\mathbf{v}_2 = (1,1)^\top$. - The activation function is Gaussian radial basis function (RBF):

$$g_i(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{v}_i\|^2}{2\sigma^2}\right).$$

Transformation Results: Applying the transformation to the inputs yields new points in the g_1-g_2 space. For example, the input $\mathbf{x} = (0,0)$ maps to $(g_1, g_2) = (1, e^{-1})$, and $\mathbf{x} = (1,1)$ maps to $(e^{-1}, 1)$. This transformation often results in the classes becoming linearly separable in the g_1–g_2 plane; plotting the four transformed points reveals that samples from different classes occupy opposite corners of the square, allowing a single linear decision boundary to separate them.

8.6 Finding the Optimal Weight Vector w

Given the transformed data $\mathbf{g}(\mathbf{x})$ and desired outputs $\mathbf{d}$, we want to find $\mathbf{w}$ that minimizes the squared error between the predicted output and the target. Let $\Phi \in \mathbb{R}^{N \times M}$ be the design matrix with entries $\Phi_{ji} = \varphi_i(\mathbf{x}_j) = G_i(\mathbf{x}_j)$. The model predicts $\hat{\mathbf{d}} = \Phi\mathbf{w}$, and the least-squares objective is

$$J(\mathbf{w}) = \|\mathbf{d} - \Phi\mathbf{w}\|^2. \tag{8.3}$$

Normal Equations for the Weights: Differentiating (8.3) with respect to $\mathbf{w}$ and setting the gradient to zero yields

$$\Phi^\top \Phi\, \mathbf{w} = \Phi^\top \mathbf{d}. \tag{8.4}$$

When $\Phi^\top\Phi$ is well conditioned, the closed-form solution is $\mathbf{w}^\star = (\Phi^\top\Phi)^{-1}\Phi^\top\mathbf{d}$; in practice we almost always add ridge regularization as described in the training section below.

Conditioning and capacity. When M is large and Gaussians overlap heavily, $\Phi^\top\Phi$ can become ill-conditioned. Ridge regularization (adding λI) stabilizes the solve and controls variance, mirroring the bias–variance trade-off from Chapter 3. Choosing M, σ, and λ together is essential for good generalization; Chapter 3's learning-curve diagnostics apply directly, and kernel methods (e.g., kernel ridge regression or SVMs) interpret the same trade-off via RBF kernels.

8.7 The Role of the Transformation Function $g(\cdot)$

The design matrix Φ is constructed by applying a nonlinear transformation $g(\cdot)$ to the input data points relative to a set of centroids $\{\mathbf{v}_i\}$. Each element of Φ is typically defined as:

$$\Phi_{ji} = g_i(\mathbf{x}_j) = g\left(\|\mathbf{x}_j - \mathbf{v}_i\|\right).$$

where $\|\cdot\|$ denotes a norm (usually Euclidean distance), and $g(\cdot)$ is a nonlinear kernel or activation function.

Two parameters characterize $g(\cdot)$:

- $\mathbf{v}_i$: the centroid or center of the i-th basis function.
- σ_i: the width or spread parameter controlling the receptive field of the basis function.

Choosing $g(\cdot)$: The choice of $g(\cdot)$ is crucial. It defines how the input space is mapped into the feature space where linear separation is possible. A common rule-of-thumb for Gaussian widths is to set σ so that neighboring centers at average spacing $\bar{r}$ overlap with height $\exp\!\big(-\bar{r}^2/(2\sigma^2)\big) \approx 0.5$–$0.7$; too small σ fragments the boundary, too large washes out locality.

8.8 Examples of Kernel Functions

1. Inverse Distance Function:

$$g(r) = \frac{1}{r+\epsilon}, \quad \epsilon > 0,$$

where $r = \|\mathbf{x} - \mathbf{v}\|$. This function decreases as the distance increases but can become unbounded near zero, potentially causing numerical instability.

2. Gaussian Radial Basis Function:

$$g(r) = \exp\left(-\frac{r^2}{2\sigma^2}\right).$$

This function is smooth, bounded, and has a clear interpretation as a localized receptive field centered at $\mathbf{v}$ with width σ. It is the most commonly used kernel in RBF networks.

Author's note: why "radial" and why a Gaussian?

An RBF unit is called *radial* because its response depends primarily on distance from a center: points at the same radius (in the chosen metric) produce the same activation. The Gaussian basis is popular because it is smooth, has a clear center, and its width parameter σ directly controls locality: large σ makes each unit "see" broadly (risking underfit), while small σ makes units highly local (risking overfit and poor conditioning). The practical art is to pick centers that cover the data and then tune σ (and ridge λ) by validation, as in Figure 31.

8.9 Interpretation of the Width Parameter σ

The parameter σ controls the spread of the basis function. Conceptually, increasing σ broadens the Gaussian bell, while decreasing σ produces a narrow spike around the centroid.

- $\sigma = 1$: The function is broad, covering a large region of the input space.
- $\sigma = 0.3$: The function is narrow and sharply peaked around the centroid.

Choosing σ appropriately is critical for the network's performance:

- If σ is too large, the basis functions overlap excessively, leading to smooth but potentially underfitting models.
- If σ is too small, the basis functions become too localized, which may cause overfitting and poor generalization.

8.10 Effect of σ on Classification Boundaries

Consider a one-dimensional dataset with two classes (e.g., red and blue points). Projecting a sample x through the Gaussian basis functions produces feature

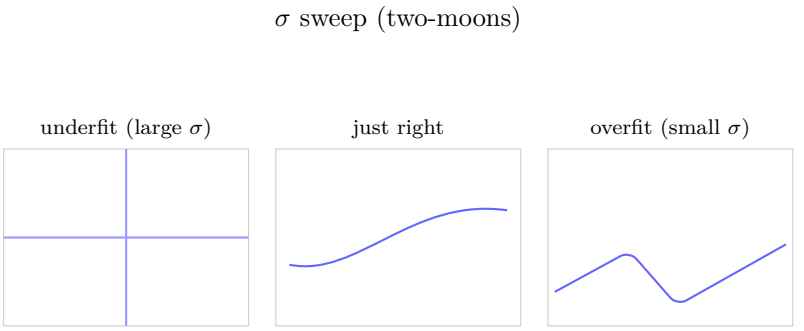

Figure 31: Schematic: How the width parameter (sigma) influences decision boundaries on a 2D toy dataset. Too-large sigma underfits, intermediate sigma captures the boundary, too-small sigma overfits with fragmented regions. Use Chapter 3's validation curves to pick model size and regularization.

activations

$$\varphi_i(x) = \exp\left(-\frac{(x - v_i)^2}{2\sigma^2}\right),$$

which serve as localized similarity measures to each centroid v_i. When σ is large, many points activate the same basis functions with comparable strength, leading to smooth decision boundaries after the linear output layer. When σ is small, only points very close to a centroid elicit large activations, yielding sharply varying boundaries that can overfit noise. Visualizing $\varphi_i(x)$ for several centroids illustrates how tuning σ controls the flexibility of the classifier.

Notation note. In this chapter we write radial basis functions as $\varphi_i(\cdot)$ and use Φ for the associated design matrix. When we need a generic kernel feature map, we use $\phi(\cdot)$ (consistent with Appendix B); when probability density functions are needed, we write them as $p(\cdot)$. This avoids overloading a single symbol.

8.11 Radial Basis Function Networks: Parameter Estimation and Training

Recall that in Radial Basis Function (RBF) networks, the hidden layer neurons compute outputs based on radial basis functions centered at certain points $\mathbf{v}_i$ with spread parameters σ_i. The output is a linear combination of these nonlinear

transformations. The key challenge is to determine the parameters:

$$\{\mathbf{v}_i, \sigma_i, w_i\}_{i=1}^{M},$$

where M is the number of hidden neurons.

Finding the Centers $\mathbf{v}_i$: A natural approach to find the centers is to use clustering algorithms on the input data. For example, if we decide to have M hidden neurons, we run a clustering algorithm (e.g., K-means) to find M centroids:

$$\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_M.$$

These centroids represent typical data points around which the radial basis functions are centered. This approach ensures that the radial basis functions cover the input space effectively.

Determining the Spread Parameters σ_i: The spread parameters control the width of each radial basis function. One can initialize all σ_i to a common value or assign different values based on the data distribution. A practical rule-of-thumb is

$$\sigma \approx \frac{d_{\max}}{\sqrt{2M}},$$

where $d_{\max}$ is the maximum pairwise distance between centers and M the number of RBF units; this ensures neighboring receptive fields overlap without collapsing to a constant function. After setting this global width, refine to per-center widths by setting each σ_i proportional to the average distance between the centroid $\mathbf{v}_i$ and its nearest neighboring centroids. Anisotropic variants scale each dimension separately but follow the same principle of matching the local density of prototypes.

Training the Output Weights w_i: Given fixed centers and spreads, the output weights w_i can be found by minimizing the squared error between the network output and the target values. The network output for an input $\mathbf{x}$ is:

$$\hat{y}(\mathbf{x}) = \sum_{i=1}^{M} w_i \varphi_i(\mathbf{x}).$$

where

$$\varphi_i(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x}-\mathbf{v}_i\|^2}{2\sigma_i^2}\right).$$

The training problem reduces to solving the linear system:

$$\min_{\mathbf{w}} \|\mathbf{y} - \Phi\mathbf{w}\|^2, \tag{8.5}$$

where $\mathbf{y}$ is the vector of target outputs and Φ is the design matrix with entries $\Phi_{ji} = \varphi_i(\mathbf{x}_j)$. When $\Phi^\top\Phi$ is well-conditioned, the ordinary least-squares solution is

$$\mathbf{w}^\star = (\Phi^\top\Phi)^{-1}\Phi^\top\mathbf{y}.$$

Dual viewpoint: RBFN vs. kernel ridge regression

Fixing the RBF centers and widths makes the hidden layer a finite basis expansion. Training restricts itself to the M coefficients $\mathbf{w}$ and resembles kernel ridge regression with a truncated basis. In the dual view, kernel ridge regression solves

$$\min_{\boldsymbol{\alpha}} \|\mathbf{y} - K\boldsymbol{\alpha}\|^2 + \lambda\boldsymbol{\alpha}^\top K\boldsymbol{\alpha},$$

where $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$ uses the same Gaussian kernel. Setting $M = N$ and letting the RBF centers coincide with the training points recovers this dual form exactly. Finite M acts like Nyström approximation: $\Phi\mathbf{w}$ projects onto a subset of kernel features.
Numerically, $\Phi^\top\Phi$ can be ill-conditioned if the bases overlap excessively or if centers cluster tightly; kernel ridge has the same issue via K.
Regularization is therefore essential: add λI before inversion,

$$\mathbf{w}^\star = (\Phi^\top\Phi + \lambda I)^{-1}\Phi^\top\mathbf{y},$$

mirroring the $\lambda\boldsymbol{\alpha}^\top K\boldsymbol{\alpha}$ term in the dual problem. Larger λ damps coefficients when σ is large (heavy overlap) or when data are noisy, while smaller λ preserves sharper fits at the cost of conditioning. Choosing λ via cross-validation keeps both primal (RBFN) and dual (kernel ridge) systems stable.

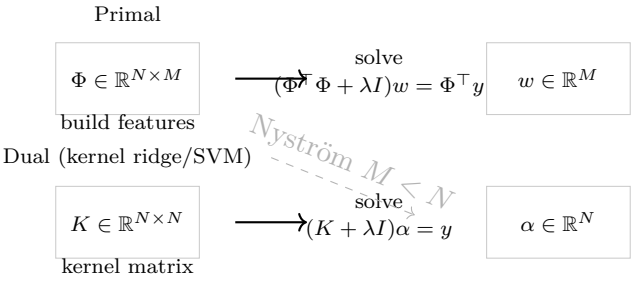

Figure 32: Schematic: Primal (finite basis) vs. dual (kernel ridge) viewpoints. Using as many centers as data points recovers the dual form; using fewer centers corresponds to a Nyström approximation. The same trade-off appears in kernel methods through the choice of kernel and effective rank.

To improve numerical stability or control model complexity, a Tikhonov (ridge) regulariser can be added,

$$\mathbf{w}_\lambda^\star = (\Phi^\top \Phi + \lambda I)^{-1} \Phi^\top \mathbf{y}, \qquad \lambda > 0,$$

or more generally one can use the Moore–Penrose pseudoinverse Φ^+ when $\Phi^\top \Phi$ is singular, yielding $\mathbf{w}^\star = \Phi^+ \mathbf{y}$. A quick dimensional sanity check is that $\Phi \in \mathbb{R}^{N\times M}$, $\mathbf{w} \in \mathbb{R}^M$, and $\mathbf{y} \in \mathbb{R}^N$; all matrix products above respect these shapes.

Iterative Optimization of σ_i and w_i: Since both σ_i and w_i affect the network output, an alternating optimization procedure can be employed:

1. Initialize σ_i (e.g., all equal or based on data heuristics).
2. Fix σ_i and find w_i by solving the linear least squares problem (8.5).
3. Fix w_i and update σ_i to minimize the error, possibly using gradient-based methods or heuristics.
4. Repeat steps 2 and 3 until convergence or error criteria are met.

Note that the spreads σ_i can be scalar or vector-valued (anisotropic), allowing different widths in each input dimension:

$$\sigma_i = [\sigma_{i1}, \sigma_{i2}, \ldots, \sigma_{id}],$$

where d is the input dimension.

Summary of the Training Algorithm:

1. Use clustering (e.g., K-means) to find centers $\mathbf{v}_i$ (or sample centers uniformly at random).
2. Set widths σ_i via a rule-of-thumb (global σ from average center spacing or per-cluster covariance).
3. Build Φ with entries $\Phi_{ji} = \varphi_i(\mathbf{x}_j)$; choose a small grid of λ values and solve $(\Phi^\top \Phi + \lambda I)\mathbf{w} = \Phi^\top \mathbf{y}$.
4. Evaluate on a validation set and pick (σ, λ, M) that minimizes validation loss; for classification, CE/hinge losses are also feasible with the same design matrix Φ.

Practical RBFN training (pseudocode)

```
Input: X, y, M, center_method=kmeans, sigma_rule, lambda_grid
Centers = center_method(X, M)
sigma = sigma_rule(Centers)
Phi = build_design_matrix(X, Centers, sigma)   # NxM
for lambda in lambda_grid:
    w_lambda = solve((Phi^T Phi + lambda I) w = Phi^T y)
    val_err[lambda] = validation_loss(Phi_val, y_val, w_lambda)
lambda_star = argmin val_err
Predict: yhat(x) = phi(x, Centers, sigma)^T w_lambda_star
```

Worked toy (classification, XOR-like). Consider four points and XOR labels

$$\mathbf{x}_1 = (0,0),\ \mathbf{x}_2 = (0,1),\ \mathbf{x}_3 = (1,0),\ \mathbf{x}_4 = (1,1), \qquad \mathbf{t} = [0,1,1,0].$$

Choose $M = 4$ centers at the data and set a global σ from the mean inter-center distance (here $\sigma \approx 0.8$). Build Φ with entries $\Phi_{ji} = \exp\big(-\|\mathbf{x}_j - \mathbf{c}_i\|^2/(2\sigma^2)\big)$ and solve $(\Phi^\top \Phi + \lambda I)\mathbf{w} = \Phi^\top \mathbf{t}$ over a small grid $\lambda \in \{10^{-4}, 10^{-3}, 10^{-2}\}$. The best λ yields a linear separator in the lifted Φ-space that perfectly classifies

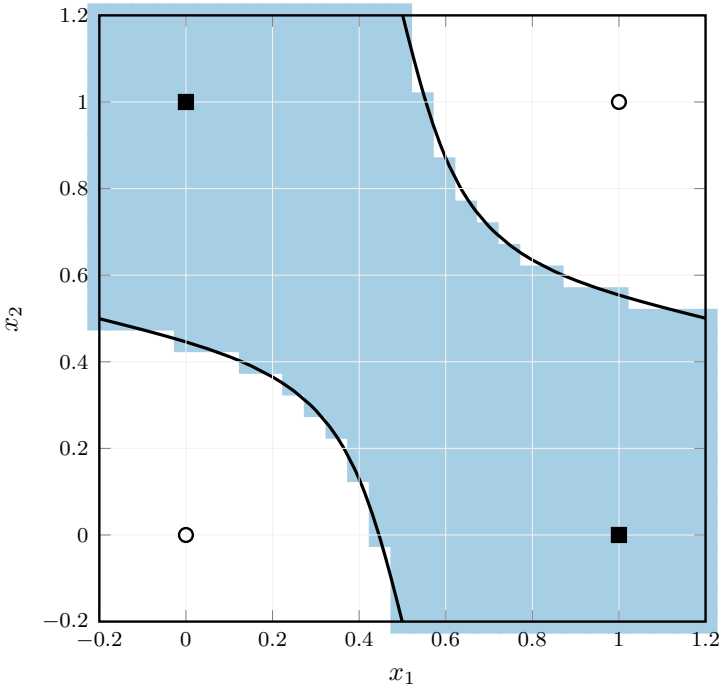

Figure 33: Schematic: RBFN decision boundary on the XOR toy for a model with 4 centers, width sigma = 0.8, and ridge lambda = 1e-3. Shading indicates the predicted class under a 0.5 threshold; the black contour marks the 0.5 boundary. Training points are overlaid (class 0: open circles; class 1: filled squares). See Figure 31 for how sigma changes this boundary.

XOR; widening σ underfits, while shrinking σ without ridge overfits via an ill-conditioned $\Phi^\top \Phi$.

8.12 Remarks on Radial Basis Function Networks

Advantages:

- **Training speed:** Once centers and spreads are fixed, training reduces to a linear least squares problem with a closed-form solution, which is computationally efficient.
- **Universal approximation:** RBF networks can approximate any continuous function on a compact domain to arbitrary accuracy given sufficient neurons, provided the centers cover the domain and the widths are chosen to avoid degeneracy (Micchelli, 1986; Park and Sandberg, 1991).

- **Interpretability:** Centers correspond to representative data points, making the network structure more interpretable.
- **Applications:** RBF networks have been successfully applied in control systems, communication systems, chaotic time series prediction (e.g., weather and power load forecasting), and decision-making tasks.
- **Flexible losses:** Squared loss is standard for regression; logistic or hinge losses pair naturally with the fixed design matrix for classification.

Disadvantages:

- **Parameter selection:** Choosing the number of neurons M, centers $\mathbf{v}_i$, and spreads σ_i is nontrivial and often requires heuristics or cross-validation.
- **Scalability:** The number of radial units required can grow quickly with input dimensionality, increasing computation and storage costs.
- **Center determination:** Identifying good centers (via clustering or other heuristics) can be computationally expensive and sensitive to noisy data.

Optional sidebar: Wiener filter refresher

Sidebar: why include Wiener filtering here? RBF networks inherit many ideas from linear estimation (projection onto bases, Gaussian kernels). The short Wiener-filter recap below is optional; feel free to skim or treat it as context that connects kernelized least-squares estimators to the localized nonlinear networks that opened this chapter.

Recall from the previous discussion that the Wiener filter aims to minimize the mean squared error (MSE) between the desired signal $d(t)$ and the filter output $y(t)$, where

$$y(t) = \mathbf{w}^T \mathbf{x}(t),$$

with $\mathbf{w}$ the filter coefficient vector and $\mathbf{x}(t)$ the input vector.

The MSE cost function is

$$J(\mathbf{w}) = \mathbb{E}\left[|d(t) - \mathbf{w}^T \mathbf{x}(t)|^2\right]. \tag{8.6}$$

To find the optimal $\mathbf{w}^\star$, we set the gradient of $J(\mathbf{w})$ with respect to $\mathbf{w}$ to

zero:

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = -2\mathbf{p} + 2\mathbf{R}\mathbf{w} = \mathbf{0},$$

where

$$\mathbf{R} = \mathbb{E}[\mathbf{x}(t)\mathbf{x}^T(t)], \quad \mathbf{p} = \mathbb{E}[d(t)\mathbf{x}(t)].$$

Solving for $\mathbf{w}$, we obtain the Wiener-Hopf equation:

$$\mathbf{R}\mathbf{w}^\star = \mathbf{p}. \tag{8.7}$$

Assuming $\mathbf{R}$ is invertible, the optimal filter coefficients are

$$\mathbf{w}^\star = \mathbf{R}^{-1}\mathbf{p}. \tag{8.8}$$

This completes the derivation of the Wiener filter solution.

Optional: interpretation and properties

Interpretation: The Wiener filter can be viewed as the linear estimator that projects the desired signal $d(t)$ onto the subspace spanned by the input vector $\mathbf{x}(t)$ in the least-squares sense.

Properties:

- **Optimality:** Minimizes the MSE among all linear filters.
- **Stationarity:** Requires knowledge of the second-order statistics $\mathbf{R}$ and $\mathbf{p}$, which are assumed stationary.
- **Causality:** The Wiener filter as derived is non-causal; causal versions require additional constraints.

Optional: frequency-domain form For stationary processes, the Wiener filter can be equivalently expressed in the frequency domain. Let $S_{xx}(\omega)$ and $S_{dx}(\omega)$ denote the power spectral density (PSD) of the input and the cross-PSD between desired and input signals, respectively. Then the frequency response of the Wiener filter is

$$H(\omega) = \frac{S_{dx}(\omega)}{S_{xx}(\omega)}. \tag{8.9}$$

This expression provides insight into the filter's behavior as a frequency-selective operator that emphasizes frequencies where the desired signal and input are strongly correlated.

Optional: why adaptive filtering comes next While the Wiener filter provides a closed-form solution, in practice the statistics $\mathbf{R}$ and $\mathbf{p}$ are often unknown or time-varying. This motivates adaptive filtering algorithms such as LMS and RLS, which iteratively approximate $\mathbf{w}^\star$ using observed data.

8.13 Preview: Unsupervised and Localized Learning

In Chapter 9, we move from supervised RBF models to unsupervised, self-organizing methods. Self-organizing maps (SOMs) and Hopfield-style associative memory discover structure in data (clusters, manifolds) without labeled targets, complementing the supervised architectures covered so far.

Key takeaways

- Localized Gaussian bases + linear readout give an interpretable nonlinear model; center/width/regularization choices control bias–variance.
- Primal RBFNs and kernel ridge regression are two views of the same estimator (full vs. truncated basis); regularization cures conditioning.
- RBFNs bridge learned-feature models (MLPs) and kernel methods (SVMs/GPs); they form a strong baseline for localized decision boundaries.

Exercises and lab ideas

- Train an RBFN on the two-moons dataset; sweep M and σ, add a small λ grid, plot validation curves and decision boundaries; report the (M, σ, λ) that minimizes validation error and discuss over/underfitting.
- Compare primal RBFN and kernel ridge regression with an RBF kernel on datasets of size $N \in \{200, 2000, 20\,000\}$; measure accuracy and runtime; note when each approach is preferable.
- Show that setting centers at all data points with $\lambda > 0$ yields the same predictions as kernel ridge regression; derive the relationship between $\mathbf{w}$ and $\boldsymbol{\alpha}$.
- Plot how validation error moves with (M, σ, λ) and link the curves back to the bias–variance discussion in Chapter 3.

Where we head next. Chapter 9 pivots from supervised objectives to competitive learning and prototype organization without labels. Chapter 10 then returns to recurrence from an energy-based perspective, setting up the later sequence-model chapters.

References. Full citations for works mentioned in this chapter appear in the book-wide bibliography.

9 Introduction to Self-Organizing Networks and Unsupervised Learning

Learning Outcomes

- Describe how competitive learning, cooperation, and annealing interact in SOM training.
- Monitor SOM quality via quantization/topographic errors and interpret U-Matrices.
- Connect SOMs to broader unsupervised techniques (clustering, dimensionality reduction) and know when to use each.

Chapter 8 offered a nonlinearity that stays close to linear algebra (basis expansion + linear solve). Here we switch to an unsupervised lens: self-organizing maps discover prototypes and organize them on a lattice without labels. The roadmap in Figure 1 highlights this as the competitive/unsupervised branch.

Design motif

Competition plus cooperation: pick a winner, then let its neighbors learn too, so the map becomes both a clustering device and a visualization.

Tiny numeric step (online update)

Input $\mathbf{x} = [0.2, 0.8]$, two map units with weights $\mathbf{w}_1 = [0.1, 0.9]$, $\mathbf{w}_2 = [0.7, 0.3]$, coordinates $\mathbf{r}_1 = [0, 0]$, $\mathbf{r}_2 = [1, 0]$, $\alpha = 0.5$, $\sigma = 1$. BMU $c = 1$ (closest to $\mathbf{x}$). Neighborhoods: $h_{11} = 1$, $h_{21} = \exp(-1/2) \approx 0.607$. Updates:

$$\mathbf{w}_1 \leftarrow [0.15,\ 0.85], \quad \mathbf{w}_2 \leftarrow [0.548,\ 0.452].$$

Even the neighbor moves toward $\mathbf{x}$, illustrating cooperation.

In this chapter, we begin our exploration of unsupervised neural networks with *Self-Organizing Maps* (SOMs), also known as Kohonen maps. Chapter 10 then studies Hopfield networks, an energy-based associative memory model. Both operate without explicit target labels, contrasting with the supervised ERM pipeline introduced in Chapter 3.

Historical intuition: two sheets and topographic neighborhoods

A useful way to picture SOMs is as two coupled "sheets": an input space and a fixed lattice of units. Each input is connected (in principle) to the whole lattice, but learning makes some regions of the lattice respond strongly (excitation) while others respond weakly (inhibition). The payoff is a *topographic* map: inputs that are far apart in the original space can end up near one another on the lattice if they are statistically similar under the features the SOM has learned.

9.1 Overview of Self-Organizing Networks

Self-organizing networks are a class of neural networks designed to discover inherent structures in input data by organizing neurons in a way that reflects the statistical properties of the data. The most prominent example is the *Self-Organizing Map* (SOM), introduced by Teuvo Kohonen. SOMs are widely used for tasks such as clustering, visualization, and dimensionality reduction.

The key characteristics of SOMs include:

- **Topology preservation:** The network maps high-dimensional input data onto a usually two-dimensional grid of neurons, preserving the topological relationships of the input space.
- **Competitive learning:** Neurons compete to become the "winner" for a given input, and only the winner and its neighbors update their weights.
- **Unsupervised learning:** No labeled outputs are required; the network self-organizes based on input similarity.

Author's note: tie SOMs back to clustering and dimensionality reduction

SOMs live in the same ecosystem as clustering and dimensionality reduction: they learn prototypes without labels and simultaneously organize those prototypes on a low-dimensional lattice. Treat the update rules as a carefully annealed clustering algorithm whose output just happens to be arranged on a grid for interpretability.

The neighborhood influence is usually controlled by a kernel (often Gaussian) whose amplitude decays with lattice distance and shrinks as training progresses,

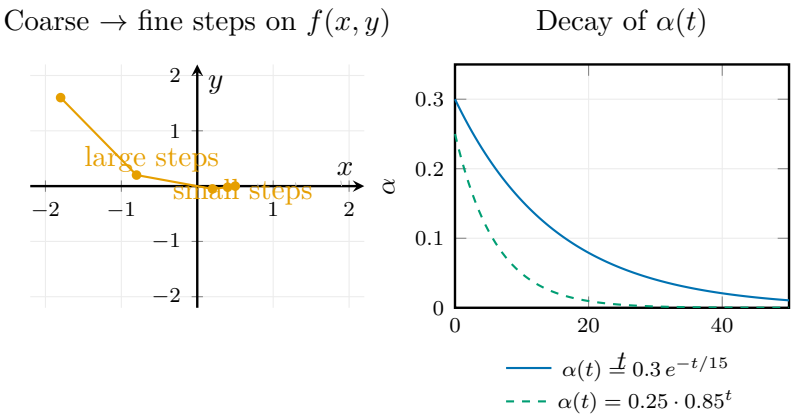

Figure 34: Schematic: Learning-rate scheduling intuition. On a smooth objective (left), large initial steps quickly cover ground and roughly align prototypes, while a decaying step-size refines the solution near convergence. Right: common exponential and multiplicative decays used in SOM training.

so early updates promote global organization while later updates refine only the closest units. Figure 34 juxtaposes these two time scales: the left panel shows why coarse early steps help traverse the energy landscape quickly, while the right panel compares two decaying learning-rate schedules commonly used when training SOMs.

Before delving into the mathematical formulation and algorithmic details of SOMs, it is important to review two foundational concepts that underpin their operation: *clustering* and *dimensionality reduction*.

9.2 Clustering: Identifying Similarities and Dissimilarities

Clustering is the process of grouping a set of objects such that objects within the same group (cluster) are more similar to each other than to those in other groups. Formally, given a dataset $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$ where each $\mathbf{x}_i \in \mathbb{R}^d$ is represented by a feature vector, the goal is to partition the data into K clusters $\{C_1, C_2, \ldots, C_K\}$ such that:

- **Intra-cluster similarity** is maximized: points within the same cluster are close to each other.
- **Inter-cluster dissimilarity** is maximized: points in different clusters are far apart.

In the classical formulation used here (e.g., for K-means), the clusters form a partition of $\mathcal{X}$: they are disjoint and their union equals the entire dataset.

Example: Consider three types of geometric shapes (triangles, circles, and squares) represented only by their feature vectors without labels. Clustering aims to group these shapes into clusters corresponding to their types based on similarity in features, even though the network does not know the labels.

K-means Clustering: A classical and widely used clustering algorithm is *K-means*, which operates as follows:

1. Initialize K cluster centroids $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_K\}$ randomly.
2. For each data point $\mathbf{x}_i$, assign it to the cluster with the nearest centroid:

$$c_i = \arg\min_k \|\mathbf{x}_i - \mathbf{v}_k\|_2, \tag{9.1}$$

   where $\|\cdot\|_2$ denotes the Euclidean norm.
3. Update each centroid as the mean of all points assigned to it:

$$\mathbf{v}_k = \frac{1}{|C_k|} \sum_{\mathbf{x}_i \in C_k} \mathbf{x}_i, \tag{9.2}$$

   where $|C_k|$ is the number of points in cluster C_k.
4. Repeat steps 2 and 3 until convergence (i.e., cluster assignments no longer change significantly).

K-means is an unsupervised learning method because it does not require labeled data; it discovers clusters purely based on feature similarity.

9.3 Dimensionality Reduction: Simplifying High-Dimensional Data

Dimensionality reduction refers to techniques that transform high-dimensional data into a lower-dimensional representation while preserving important structural properties, such as pairwise distances, variance, or neighborhood relationships. This is crucial for:

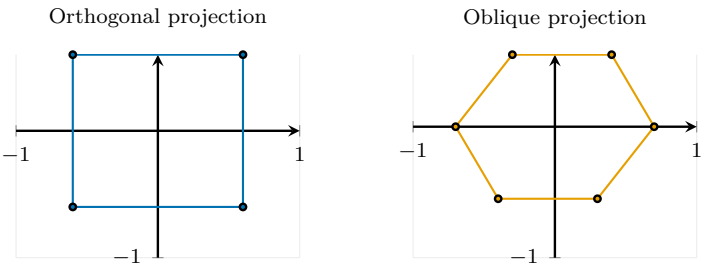

Figure 35: Schematic: Classical MDS intuition. Projecting a cube onto a plane via an orthogonal map yields a square (left), whereas an oblique projection along a body diagonal produces a hexagon (right). The local adjacency of vertices is preserved even though metric structure is distorted.

- **Visualization:** Humans can easily interpret data in two or three dimensions.
- **Computational efficiency:** Reducing dimensions can simplify subsequent processing.
- **Noise reduction:** Eliminating irrelevant or redundant features.

Example: Consider a three-dimensional cube. Depending on its orientation, a linear projection (matrix multiplication by $P : \mathbb{R}^3 \to \mathbb{R}^2$ with matrix representation in $\mathbb{R}^{2\times 3}$) onto a two-dimensional plane can look like different shapes: a square arises from an orthogonal projection onto a face, whereas a hexagon appears under an oblique projection along a body-diagonal. This highlights that while the combinatorial adjacency (which vertices are connected) is preserved under such a projection, Euclidean lengths and angles are inevitably distorted. Figure 35 illustrates these two views.

Challenges: Reducing dimensions inevitably leads to some loss of information. The goal is to minimize this loss while achieving a more tractable representation.

Common Techniques: Principal Component Analysis (PCA) is a linear method that preserves orthogonal directions of maximum variance (the eigenvectors of the covariance matrix), while classical Multidimensional Scaling (MDS) reconstructs an embedding by double-centering a squared-distance

matrix ($B = -\frac{1}{2}JD^{(2)}J$, where $D_{ij}^{(2)} = \|\mathbf{x}_i - \mathbf{x}_j\|_2^2$ and $J = I - \frac{1}{n}\mathbf{1}\mathbf{1}^\top$ is the centering matrix) and performing eigen-decomposition so that Euclidean pairwise distances are approximated as closely as possible. Methods such as t-SNE or UMAP provide nonlinear embeddings that emphasize local neighborhoods but typically do not preserve global distances. Self-Organizing Maps also serve as a nonlinear dimensionality reduction technique by mapping high-dimensional inputs onto a low-dimensional lattice while preserving neighborhood relationships among data points, albeit on a discrete grid rather than a continuous embedding.

9.4 Dimensionality Reduction and Feature Mapping

Recall from the previous discussion that dimensionality reduction aims to map a high-dimensional feature space into a lower-dimensional representation while preserving as much information as possible. This is crucial in many applications such as image processing, speech recognition, and pattern analysis, where the original data may have many correlated or redundant features yet the geometric relationships (distances, variance directions, neighborhoods) must remain meaningful.

For example, consider a face represented by multiple features: eyes, nose, mouth, ears, shape of the face, etc. If we want to reduce this to three dimensions, we must carefully choose which features to combine or discard so that the essential characteristics of the face remain recognizable. A naive reduction that drops important features arbitrarily will result in poor representation.

Depending on the application, the map may be linear (a projection as in PCA) or nonlinear (a learned embedding as in t-SNE or SOM; note that SOMs produce a discrete lattice embedding rather than a continuous Euclidean embedding). The goal is to find a mapping

$$f : \mathbb{R}^n \to \mathbb{R}^m, \quad m < n,$$

such that the new feature vector $\mathbf{y} = f(\mathbf{x})$ retains the salient structure of $\mathbf{x}$, for example by approximately preserving pairwise distances, nearest neighbors, or dominant variance directions. Modern algorithms discover these combinations automatically from data, often in an unsupervised manner (PCA, t-SNE, SOM). Supervised (e.g., Linear Discriminant Analysis) and semi-supervised variants

also exist, where label information guides f, rather than relying on manual feature selection. For instance, PCA derives f analytically via eigen-decomposition of the covariance matrix, whereas t-SNE and SOM learn f iteratively from data.

9.5 Self-Organizing Maps (SOMs): Introduction

Self-Organizing Maps (SOMs), also known as Kohonen maps, provide a powerful approach to unsupervised learning that combines clustering and dimensionality reduction. Unlike supervised neural networks, SOMs learn without explicit target outputs or labels. Instead, they discover the underlying structure of the input data by organizing neurons in a topological map.

SOM at a glance

Objective: Learn prototype vectors $\mathbf{w}_i$ arranged on a low-dimensional lattice such that nearby neurons represent nearby regions of the input space (topographic mapping).
Key hyperparameters: Map size and topology, initial learning rate $\alpha(0)$, neighborhood width $\sigma(0)$ and their decay schedules, distance metric (typically squared Euclidean).
Defaults: Start with a 2D rectangular grid, squared Euclidean distance, exponentially decaying $\alpha(t)$ and $\sigma(t)$, and a number of neurons comparable to or slightly larger than the expected number of clusters.
Common pitfalls: Too small a map (forcing unrelated inputs to share neurons), overly fast decay of $\alpha(t)$ or $\sigma(t)$ (freezing the map early), and interpreting the lattice as metric when it is only approximately topology-preserving.

Historical Context The concept of SOMs traces back to early models of self-organizing topographic maps, such as the two-sheet formulation of Willshaw and von der Malsburg (1976). Teuvo Kohonen later formalized and popularized the algorithmic framework in Kohonen (1982) (see also Kohonen, 2001).

Basic Architecture Conceptually, the SOM consists of two stages:

- **Input layer:** A vector $\mathbf{x} \in \mathbb{R}^n$ representing the input features.
- **Output layer (map):** A usually two-dimensional grid of units (neurons).

Each neuron i is assigned a fixed coordinate vector $\mathbf{r}_i = [u_i, v_i]^\top$ with $u_i, v_i \in \mathbb{Z}$ together with a weight vector $\mathbf{w}_i \in \mathbb{R}^n$. The coordinates $\mathbf{r}_i$ determine geometric proximity on the lattice and are used by the neighborhood function (Section 9.16).

Each output neuron therefore possesses a weight vector of the same dimensionality as the input, so evaluating the match between an input and the map amounts to comparing the input against every stored prototype. The neurons then compete; the closest (best matching) unit "wins" and its neighbors are allowed to adapt by nudging their weight vectors toward the input, while distant units remain unchanged during that update. The resulting organization produces a discrete map that preserves qualitative ordering; it approximates the topology of the input space without providing a continuous Euclidean embedding.

Key Concept: Topographic Mapping The fundamental idea is that inputs that are similar in the original space will activate output units that are close to each other on the map. This preserves the topological relationships of the input data in the reduced-dimensional output space.

Formally, if $\mathcal{N}_\epsilon(\mathbf{x}) = \{\mathbf{z} \mid \|\mathbf{z} - \mathbf{x}\|_2 < \epsilon\}$ denotes an Euclidean ϵ-neighborhood of an input vector, the SOM training procedure aims to ensure that the image of this neighborhood under the map lies within a small neighborhood of the BMU on the lattice. In practice the preservation is approximate (see, e.g., Kohonen, 2001 for discussion), but it is sufficient to maintain qualitative ordering of regions in the input manifold.

For example, two inputs $\mathbf{x}_1$ and $\mathbf{x}_2$ that are close in $\mathbb{R}^n$ will select best matching units whose lattice locations $\mathbf{r}_i$ and $\mathbf{r}_j$ are neighbors on the output grid. This spatial organization is what makes SOMs particularly useful for visualization and clustering.

9.6 Conceptual Description of SOM Operation

1. **Initialization:** The weight vectors $\mathbf{w}_i$ are initialized, often randomly or by sampling from the input space.

2. **Competition:** For a given input $\mathbf{x}$, find the best matching unit (BMU)

or winning neuron:

$$c = \operatorname*{argmin}_i \|\mathbf{x} - \mathbf{w}_i\|_2^2, \tag{9.3}$$

that is, the BMU index c minimizes the squared Euclidean distance between $\mathbf{x}$ and the candidate prototype $\mathbf{w}_i$. Minimizing the squared distance yields the same winner as minimizing the unsquared norm but streamlines gradient derivations, so we retain the squared form for consistency with later update rules. Here $\|\cdot\|_2$ denotes the Euclidean norm unless explicitly stated otherwise. Euclidean distance is the default choice because it yields particularly simple gradient expressions for the update rule (9.4), but alternatives such as Mahalanobis distance (for anisotropic covariance structures) or cosine-based measures (e.g., the cosine distance $d_{\cos}(\mathbf{x}, \mathbf{w}_i) = 1 - \frac{\mathbf{x}^\top \mathbf{w}_i}{\|\mathbf{x}\|_2 \, \|\mathbf{w}_i\|_2}$) can be used; the metric must be chosen to reflect the notion of similarity relevant to the application. Throughout this section we denote the best matching unit (BMU) by the index c; alternative notations such as $j^\star$ or $i^\star$ in the literature refer to the same winning neuron.

3. **Cooperation:** Define a neighborhood function $h_{ci}(t)$ that determines the degree of influence the BMU has on its neighbors in the output grid. This function decreases with the distance between neurons c and i on the map and with time t.

4. **Adaptation:** Update the weight vectors of the BMU and its neighbors to move closer to the input vector:

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + \alpha(t) h_{ci}(t) \big(\mathbf{x} - \mathbf{w}_i(t)\big), \tag{9.4}$$

where $\alpha(t)$ is the learning rate, which decreases over time, and the effective width of $h_{ci}(t)$ likewise shrinks so that large-scale ordering occurs early and fine-tuning occurs later (see Section 9.16).

This iterative process causes the map to self-organize, with neurons specializing to represent clusters or features of the input space.

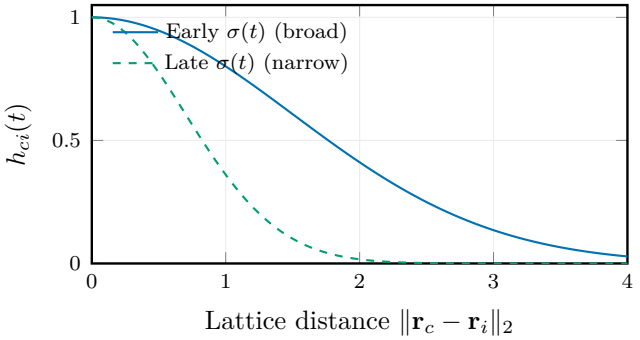

Figure 36: Schematic: Gaussian neighborhood weights in SOM training. Early iterations use a broad kernel so many neighbors adapt; later iterations shrink the neighborhood width sigma(t) so only units near the BMU update.

9.7 Mathematical Formulation of SOM

Let the input space be $\mathcal{X} \subseteq \mathbb{R}^n$, and the output map be a lattice of neurons indexed by i, each with weight vector $\mathbf{w}_i \in \mathbb{R}^n$.

Best Matching Unit (BMU) Given an input $\mathbf{x}$, the BMU is found by minimizing the squared distance:

$$c = \arg\min_i \|\mathbf{x} - \mathbf{w}_i\|_2^2.$$

Neighborhood Function A common choice for the neighborhood kernel is the Gaussian function

$$h_{ci}(t) = \exp\left(-\frac{\|\mathbf{r}_c - \mathbf{r}_i\|^2}{2\sigma^2(t)}\right), \tag{9.5}$$

where $\mathbf{r}_i$ denotes the lattice coordinates of neuron i and $\sigma(t)$ is the neighborhood radius that decreases monotonically with t. Early in training $\sigma(t)$ is large, encouraging broad cooperation; as $\sigma(t)$ shrinks, only neurons near the BMU continue to adapt (Figure 36).

9.8 Kohonen Self-Organizing Maps (SOMs): Network Architecture and Operation

Building on the inspiration from perceptrons, Kohonen Self-Organizing Maps (SOMs) introduce a distinctive neural network architecture designed for unsupervised learning and feature mapping. Unlike classical supervised networks, SOMs aim to discover the underlying structure of the input data by organizing neurons in a fixed, usually low-dimensional, lattice.

Network Structure

- **Input layer:** The input vector $\mathbf{x} \in \mathbb{R}^n$ represents the feature space, where n is the input dimension.
- **Output layer (map):** A fixed lattice of neurons arranged in a low-dimensional grid, e.g., a 6×4 or 3×3 grid, independent of the input dimension.
- **Connectivity:** Each neuron in the output layer is fully connected to all input components via a weight vector $\mathbf{w}_i \in \mathbb{R}^n$, where i indexes the neuron.

Mapping and Competition The SOM maps the high-dimensional input $\mathbf{x}$ to a single neuron in the output lattice that best represents the input. This is achieved by measuring the similarity between $\mathbf{x}$ and each neuron's weight vector $\mathbf{w}_i$. The neuron with the highest similarity (or equivalently, the smallest distance) is declared the *winner*.

Formally, the winning neuron c for input $\mathbf{x}$ is given by

$$c = \arg\min_i \|\mathbf{x} - \mathbf{w}_i\|_2^2, \tag{9.6}$$

where $\|\cdot\|_2$ denotes the Euclidean norm. Squaring the norm leaves the minimizer unchanged ($\arg\min_i \|\mathbf{x} - \mathbf{w}_i\|_2 = \arg\min_i \|\mathbf{x} - \mathbf{w}_i\|_2^2$), but simplifies derivatives in the subsequent learning rule. Alternative similarity metrics (e.g., cosine distance) can replace $\|\mathbf{x} - \mathbf{w}_i\|_2$ when appropriate.

Weight Update Rule Only the winning neuron and its neighbors in the lattice update their weights to better represent the input. This competitive

learning rule can be expressed as

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + \alpha(t)\, h_{ci}(t)\, \big(\mathbf{x}(t) - \mathbf{w}_i(t)\big), \tag{9.7}$$

where the learning rate symbol matches the one used in the conceptual outline,

- $\alpha(t)$ is the learning rate at iteration t,
- $h_{ci}(t)$ is the neighborhood function centered on the winning neuron c, typically a Gaussian kernel that decreases with the lattice distance between neurons c and i (see Equation (9.5)).

This update rule ensures that the winning neuron and its neighbors move closer to the input vector, preserving topological relationships in the input space. Intuitively, simultaneous adaptation of the BMU and its nearby units keeps neighboring weight vectors in similar regions of the input space, so the lattice retains the ordering of the data manifold.

9.9 Example: SOM with a 3×3 Output Map and 4-Dimensional Input

Consider a SOM with the following specifications:

- Input dimension: $n = 4$, so each input vector is $\mathbf{x} = [x_1, x_2, x_3, x_4]^T$.
- Output lattice: 3×3 grid, totaling 9 neurons indexed $i = 1, \ldots, 9$.
- Each neuron i has a weight vector $\mathbf{w}_i \in \mathbb{R}^4$.

Feedforward Computation For a given input $\mathbf{x}$, each neuron computes a similarity score. Two common choices are:

$$y_i = \mathbf{w}_i^\top \mathbf{x} \qquad \text{(dot-product similarity)}, \tag{9.8}$$

$$d_i = \|\mathbf{x} - \mathbf{w}_i\|_2^2 \qquad \text{(squared Euclidean distance)}. \tag{9.9}$$

In both expressions $\mathbf{w}_i$ and $\mathbf{x}$ are column vectors, so $\mathbf{w}_i^\top \mathbf{x}$ is a scalar similarity score while d_i computes the squared Euclidean distance.

When using dot products we select the neuron with the maximum y_i; when using distances we equivalently select the neuron with the minimum d_i (or the

maximum of $-d_i$):

$$c = \begin{cases} \arg\max_i y_i, & \text{if similarities are measured via (9.8),} \\ \arg\min_i d_i, & \text{if distances are used as in (9.9).} \end{cases}$$

Weight Initialization and Update Weights $\mathbf{w}_i$ are typically initialized randomly or sampled from the input distribution. During training, for each input $\mathbf{x}$, the winning neuron c and its neighbors update their weights according to (9.7).

Illustration

- Suppose the input $\mathbf{x}$ is presented.
- Compute $y_i = \mathbf{w}_i^T \mathbf{x}$ for all neurons i.
- Identify the winning neuron c with the highest y_i.
- Update $\mathbf{w}_c$ and neighboring weights $\mathbf{w}_i$ using (9.7).

This process repeats over many inputs, gradually organizing the map such that neighboring neurons respond to similar inputs, effectively performing a topology-preserving dimensionality reduction.

The lattice coordinates $\mathbf{r}_i \in \mathbb{Z}^2$ introduced for the neighborhood kernel serve as the geometry of the output grid; distances such as $\|\mathbf{r}_i - \mathbf{r}_c\|_2$ determine how strongly each neuron responds when c wins. Broad kernels (large $\sigma(t)$) encourage global ordering early in training, whereas shrinking $\sigma(t)$ confines adaptation to local neighborhoods so that fine-grained structure emerges. Alternative kernel shapes (e.g., Epanechnikov, bubble) can be used, though Gaussians provide smooth decay and convenient derivatives.

SOM training is typically stochastic: each input triggers an update, so the map continuously refines prototypes as data arrive. Batch variants exist, but online updates capture streaming data and mirror Kohonen's original algorithm. Initialization also affects convergence; besides random sampling, practical systems often initialize weights along leading principal components to align the lattice orientation with the data manifold.

9.10 Key Properties of Kohonen SOMs

- **Fixed output dimension:** The lattice size is a design choice specified a priori and does not automatically scale with the input dimension.
- **Winner-takes-all competition:** Only the best matching unit and its neighbors adapt their weights, encouraging topological ordering.
- **Neighborhood cooperation:** Updating neighboring neurons enforces smooth transitions across the map.

9.11 Winner-Takes-All Learning and Weight Update Rules

Recall that in competitive learning networks, the neuron with the highest discriminant value for a given input $\mathbf{x}$ is declared the *winner*. This subsection analyzes the classical *winner-takes-all* (WTA) principle in which only the winning neuron updates its weights, while all others remain unchanged. In the SOM setting discussed earlier, a softened variant is used in which the winner and its lattice neighbors update together.

Discriminant Function and Similarity Measures The discriminant value for neuron j is typically computed from a similarity or distance measure between the input $\mathbf{x}$ and the neuron's weight vector $\mathbf{w}_j$. Two common formulations are:

- **Maximizing similarity:**

$$g_j(\mathbf{x}) = \mathbf{w}_j^\top \mathbf{x}.$$

  where a higher inner product indicates greater similarity.

- **Minimizing distance:**

$$d_j(\mathbf{x}) = \|\mathbf{x} - \mathbf{w}_j\|_2^2.$$

  where a smaller Euclidean distance indicates greater similarity.

While both are valid, minimizing the Euclidean distance is often preferred for weight updates because it leads to more tractable learning rules.

Weight Update Rule Once the winning neuron c is identified, its weight vector $\mathbf{w}_c$ is updated to better represent the input $\mathbf{x}$. The general update rule is:

$$\mathbf{w}_c(t+1) = \mathbf{w}_c(t) + \Delta\mathbf{w}_c(t). \tag{9.10}$$

where t indexes the iteration or training cycle and $\Delta\mathbf{w}_c(t) = \mathbf{w}_c(t+1) - \mathbf{w}_c(t)$.

The increment $\Delta\mathbf{w}_c(t)$ is chosen to reduce the distance between $\mathbf{w}_c$ and $\mathbf{x}$, but not to make them identical immediately. This is because:

- Multiple inputs $\mathbf{x}$ may be represented by the same neuron.
- Immediate convergence to a single input would prevent generalization.

Hence, the update is typically proportional to the difference between $\mathbf{x}$ and $\mathbf{w}_c$:

$$\Delta\mathbf{w}_c(t) = \alpha(t)\left(\mathbf{x} - \mathbf{w}_c(t)\right). \tag{9.11}$$

where $\alpha(t) \in [0, 1)$ is the *learning rate* at iteration t. The learning rate controls the step size so that $\mathbf{w}_c$ moves toward $\mathbf{x}$ gradually rather than collapsing to it in a single update.

Learning Rate Schedule The learning rate $\alpha(t)$ controls the magnitude of weight updates. It typically decreases over time to ensure convergence and stability:

$$\alpha(t+1) \leq \alpha(t), \quad \lim_{t\to\infty} \alpha(t) = 0.$$

This schedule allows large adjustments early in training (rapid learning) and fine-tuning later (stabilization). Practitioners often start with $\alpha(0)$ in the range 0.05–0.5 and decay it toward 10^{-3} or smaller so that updates remain responsive initially but become conservative as the map stabilizes.

Summary of the Competitive Learning Algorithm

1. Initialize weights $\mathbf{w}_j(0)$ randomly or heuristically.
2. For each input $\mathbf{x}$:

(a) Compute discriminant functions $g_j(\mathbf{x})$ or distances $d_j(\mathbf{x})$.

(b) Select winning neuron:

$$c = \arg\max_j g_j(\mathbf{x}) \quad \text{or} \quad c = \arg\min_j d_j(\mathbf{x})$$

(c) Update the winning neuron's weights using (9.10) and (9.11).

3. Decrease learning rate $\alpha(t)$ according to schedule.

4. Repeat until convergence or maximum iterations reached.

9.12 Numerical Example of Competitive Learning

Consider a simple example with:

- Four input vectors $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4 \in \mathbb{R}^4$.
- A competitive layer with three neurons (clusters).
- Initial learning rate $\alpha(0) = 0.3$ with multiplicative decay $\alpha(t) = 0.3 \times 0.5^t$ (ensuring $\alpha(t) > 0$).
- No neighborhood function (i.e., only the winner updates).

Initial Weights The initial weights $\mathbf{w}_j(0)$ for neurons $j = 1, 2, 3$ are:

$$\mathbf{W}(0) = \begin{bmatrix} 0.2 & 0.3 & 0.5 & 0.1 \\ 0.2 & 0.3 & 0.1 & 0.4 \\ 0.3 & 0.5 & 0.2 & 0.3 \end{bmatrix}$$

where row j contains the initial weight vector $\mathbf{w}_j(0)$ for neuron $j = 1, 2, 3$.

9.13 Winner-Takes-All Learning Recap

Recall from the previous discussion that in the Winner-Takes-All (WTA) learning scheme, for each input vector $\mathbf{x}$, we compute the similarity (or distance) between $\mathbf{x}$ and each neuron's weight vector $\mathbf{w}_j$. The neuron c with the minimum distance (or maximum similarity) is declared the winner:

$$c = \arg\min_j \|\mathbf{x} - \mathbf{w}_j\|_2^2. \tag{9.12}$$

Only the weights of the winning neuron are updated according to:

$$\mathbf{w}_c(t+1) = \mathbf{w}_c(t) + \alpha(t)\,(\mathbf{x} - \mathbf{w}_c(t))\,, \tag{9.13}$$

where $\alpha(t)$ is the learning rate (constant or decaying). In the full SOM update of (9.7), this increment is additionally scaled by the neighborhood kernel $h_{ci}(t)$ so that only units with lattice coordinates $\mathbf{r}_i$ near the BMU location $\mathbf{r}_c$ receive appreciable adjustments.

This process is repeated for each input in the training set, and multiple epochs are run with a gradually decreasing α until convergence.

Practical considerations In both SOMs and WTA networks, input vectors are commonly normalized (e.g., zero mean and unit variance) so that distance comparisons are meaningful. Training is typically terminated when weight changes fall below a small threshold or after a prescribed number of epochs.

9.14 Regularization and Monitoring During SOM Training

Even though SOMs are inherently unsupervised, their training dynamics still benefit from the same regularization heuristics used in supervised settings. Two complementary diagnostics are especially useful in practice.

Bias–variance view. Increasing the lattice resolution or keeping the kernel width large for too long can overfit local noise. Figure 37 visualizes the familiar U-shaped trade-off: the left regime underfits (high bias), whereas the right regime yields jagged maps (high variance).

Loss-landscape smoothing. Adding small cooperative penalties (e.g., weight decay between neighbors) produces smoother loss contours and accelerates convergence, as sketched in Figure 38. The penalty discourages neighboring prototypes from diverging and keeps the map topologically ordered.

Quantization vs. information preservation. Classical SOM optimizes a topology-preserving vector quantization objective; it does not include cross-entropy terms. Modern variants sometimes introduce *auxiliary* regularizers

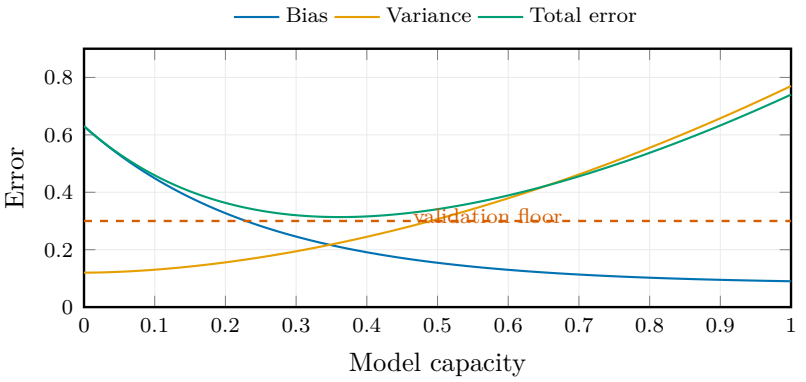

Figure 37: Schematic: Bias–variance trade-off when sweeping SOM capacity (number of units or kernel width). The optimum appears near the knee where bias and variance intersect.

to encourage codebook utilization (e.g., entropy penalties on assignment histograms) or draw analogies to VQ-VAE. Monitoring both quantization error and an entropy-style regularizer, as in Figure 39, helps reveal when the map is collapsing to a few units or when density variations are no longer represented faithfully.

Quantization vs. topographic error. Given data points $\{\mathbf{x}_i\}$ and best-matching units $b_i = \mathrm{BMU}(\mathbf{x}_i)$, the *quantization error* is

$$\mathrm{QE} = \frac{1}{N}\sum_{i=1}^{N} \left\|\mathbf{x}_i - \mathbf{w}_{b_i}\right\|_2,$$

which measures reconstruction fidelity. The *topographic error* is the fraction of inputs whose first- and second-best BMUs are not adjacent on the lattice (default: 4-neighbor connectivity), capturing topology preservation. Both metrics reappear in later figures; we monitor QE for representation quality and TE for magnification distortions.

Batch SOM in practice

Online SOM updates one sample at a time: pick a best-matching unit (BMU), nudge it and its neighbors, move on. Batch SOM instead aggregates responsibilities across a dataset (or mini-batch) before shifting

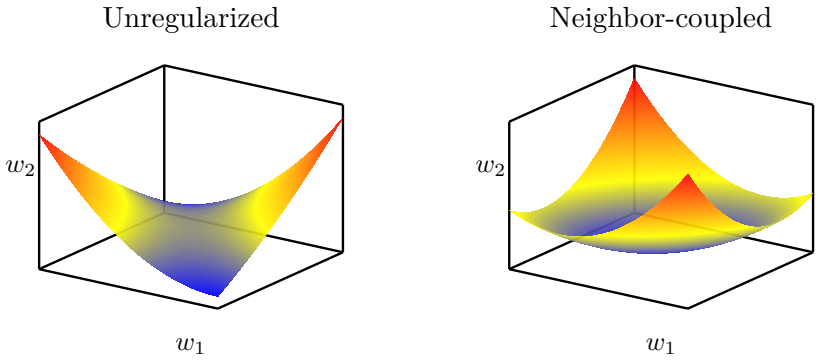

Figure 38: Schematic: Regularization smooths the loss surface. Coupling neighboring prototypes (right) yields wider, flatter basins than the jagged unregularized landscape (left).

prototypes:

$$h_{j,i}(t) = \kappa\big(\mathrm{dist}(j, b(i)); \sigma_t\big),$$
$$\mathbf{w}_j^{(t+1)} = \frac{\sum_i h_{j,i}(t)\, \mathbf{x}_i}{\sum_i h_{j,i}(t)}.$$

Key differences:

- **Deterministic passes.** Batch updates remove stochastic noise and converge in fewer epochs on static datasets, making results reproducible (useful for dashboards/visual analytics).
- **Parallelism.** Computations collapse to matrix ops (compute BMUs, accumulate weighted sums), so GPUs/CPUs can process large mini-batches efficiently.
- **Streaming trade-off.** Online updates remain preferable when data arrive continuously or when you need the map to adapt mid-stream; batch SOM suits offline datasets.

Most modern SOM libraries expose both modes, so choose the update rule that matches your data pipeline and stability requirements.

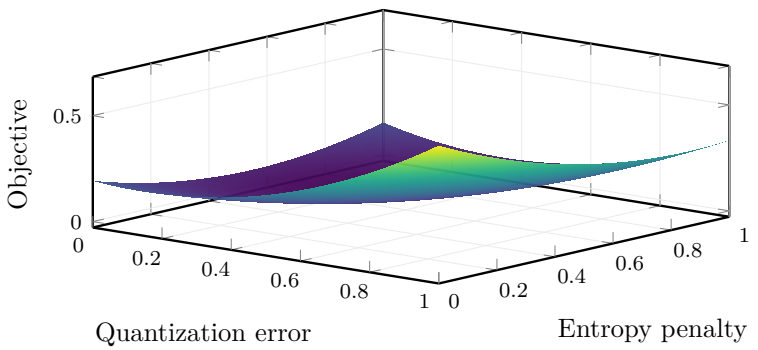

Figure 39: Schematic: Quantization error combined with an entropy-style regularizer (modern SOM variant; for example, a negative sum of p log p over unit usage). Valleys arise when prototypes cover the space evenly; ridges highlight collapse or poor topological preservation.

Stopping criteria. Because stochastic updates can eventually increase topographic error, it is standard to stop training once a moving-average validation curve plateaus. Figure 40 shows the canonical trend: fast initial improvement followed by saturation.

Key takeaways

- SOMs perform topology-preserving vector quantization on a discrete lattice.
- A shrinking neighborhood and decaying learning rate drive coarse-to-fine organization.
- U-Matrices and quantization/topographic errors are practical diagnostics for convergence.

9.15 Limitations of Winner-Takes-All and Motivation for Cooperation

While WTA is simple and effective for clustering, it has some limitations:

- Only one neuron updates per input, which can lead to slow convergence.
- The hard competition ignores relationships among neighboring neurons.
- The resulting clusters correspond to hard assignments, so boundaries between codebook vectors are sharp with little smoothing across neighboring

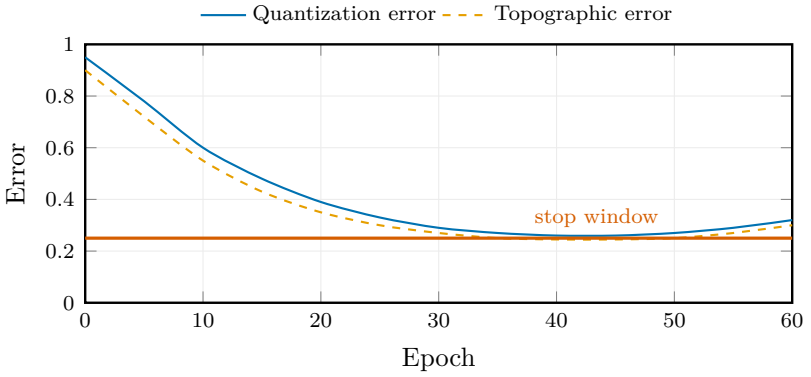

Figure 40: Schematic: Validation curves used to identify an early-stopping knee. When both quantization and topographic errors flatten (shaded band), further training risks map drift.

neurons.

The geometric effect of these limitations is easiest to see in Figure 41: the left panel shows the brittle Voronoi partitions created by a strict winner-takes-all rule, whereas the right panel demonstrates how shrinking the neighborhood kernel produces softer responsibilities and smoother maps.

To address these issues, the concept of *cooperation* among neurons is introduced. Instead of a single winner neuron updating its weights, a neighborhood of neurons around the winner also update their weights, albeit to a lesser extent. This idea leads to smoother mappings and better topological ordering.

9.16 Cooperation in Competitive Learning

Neighborhood Concept Consider the output layer arranged in a 2D grid (or lattice) of neurons. For each input $\mathbf{x}$, after determining the winning neuron c, we define a neighborhood $\mathcal{N}(c)$ consisting of neurons close to c in the output space. In practice the neighborhood weight is supplied by the kernel $h_{jc}(t)$ of (9.5), which is positive for units inside the neighborhood (and decays with the lattice distance $\|\mathbf{r}_j - \mathbf{r}_c\|$) and zero for units far away.

The neighborhood size typically shrinks over time during training, starting large to encourage global ordering and gradually reducing to fine-tune local details.

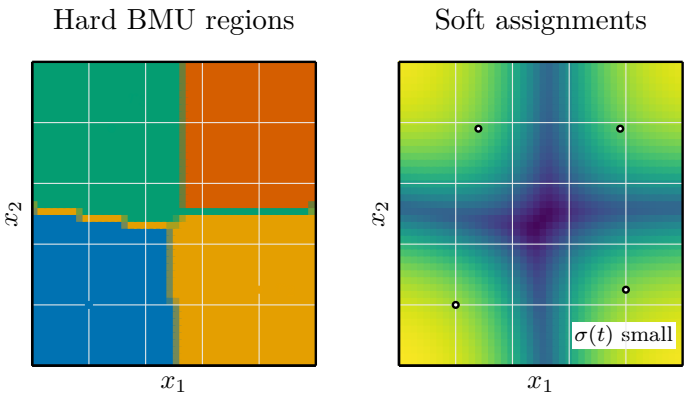

Figure 41: Schematic: Voronoi-like regions induced by SOM prototypes (left) and the corresponding softmax confidence after shrinking the neighborhood kernel (right). Softer updates blur the decision frontiers and reduce jagged mappings between adjacent neurons.

Weight Update with Neighborhood Cooperation The lattice structure and how the best matching unit (BMU) influences nearby neurons are visualized in Figure 42. The U-Matrix on the right provides a quick diagnostic for cluster boundaries during training.

The weight update rule generalizes to:

$$\mathbf{w}_j(t+1) = \mathbf{w}_j(t) + \alpha(t)\, h_{jc}(t)\, (\mathbf{x} - \mathbf{w}_j(t))\,, \tag{9.14}$$

where

- $h_{jc}(t)$ is the *neighborhood function* that quantifies the degree of cooperation between neuron j and the winner c.
- $\alpha(t)$ is the learning rate at time t.

The neighborhood function satisfies:

$$h_{jc}(t) = \begin{cases} 1, & j = c \\ \in (0,1), & j \in \mathcal{N}(c), j \neq c \\ 0, & \text{otherwise} \end{cases}$$

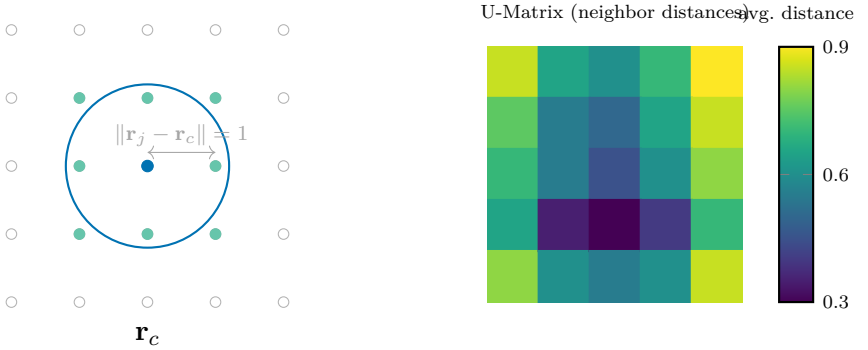

Figure 42: Schematic: Left: a 5-by-5 SOM lattice with best matching unit (blue) and neighbors within the Gaussian-kernel radius (green). Right: a toy U-Matrix (colormap chosen to remain interpretable in grayscale) showing average distances between neighboring codebook vectors; higher distances indicate likely cluster boundaries.

Gaussian Neighborhood Function A common choice for $h_{jc}(t)$ is a Gaussian function based on the distance between neurons j and c on the output lattice:

$$h_{jc}(t) = \exp\left(-\frac{\|\mathbf{r}_j - \mathbf{r}_c\|^2}{2\sigma^2(t)}\right), \tag{9.15}$$

where

- $\mathbf{r}_j$ and $\mathbf{r}_c$ are the coordinates of neurons j and c on the output grid.
- $\sigma(t)$ is the neighborhood radius (width) at time t, which decreases over training.

This function ensures that neurons closer to the winner receive larger updates, while distant neurons are updated less or not at all.

Interpretation The cooperative update encourages neighboring neurons to become sensitive to similar inputs, thereby preserving topological relationships in the input space. This is the key principle behind Self-Organizing Maps (SOMs).

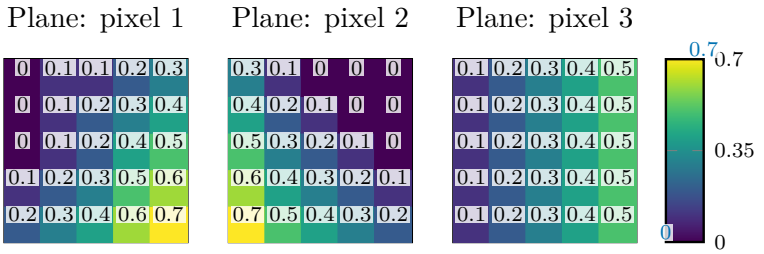

Figure 43: Schematic: Component planes for three features on a trained SOM (toy data). Each plane maps one feature's value across the map; aligned bright/dark regions across planes reveal correlated features, complementing the U-Matrix in Figure 42. Interpret brightness comparatively within a plane rather than as an absolute calibrated scale.

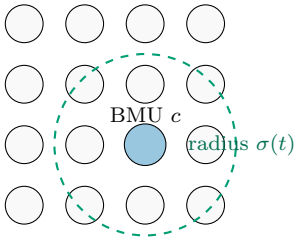

Figure 44: Schematic: SOM lattice with the best-matching unit (BMU) highlighted in blue and a dashed neighborhood radius indicating which prototype vectors receive cooperative updates.

9.17 Example: Neighborhood Update Illustration

Suppose the output neurons are arranged in a 2D lattice as shown schematically in Figure 44, where each neuron is indexed by its grid coordinates. For an input $\mathbf{x}$, neuron c wins. The neighborhood $\mathcal{N}(c)$ might include neurons within a radius σ around c.

Each neuron j in $\mathcal{N}(c)$ updates its weight vector according to (9.14), with the magnitude of update modulated by $h_{jc}(t)$.

9.18 Summary of Cooperative Competitive Learning Algorithm

1. Present an input vector and identify the winning neuron using the discriminant function.

2. Update the winning neuron's weights and those of its neighbors according to the cooperative rule.

3. Decrease the learning rate and neighborhood radius according to the annealing schedule.

4. Repeat for all inputs until the map stabilizes or a maximum number of epochs is reached.

9.19 Wrapping Up the Kohonen Self-Organizing Map (SOM) Derivations

We conclude our derivation and discussion of the Kohonen Self-Organizing Map (SOM) learning algorithm by summarizing the key components and their evolution during training.

Recall the weight update rule for neuron j at time step t:

$$\Delta \mathbf{w}_j(t) = \alpha(t)\, h_{j,c}(t)\, [\mathbf{x}(t) - \mathbf{w}_j(t)]\,. \tag{9.16}$$

where:

- $\mathbf{x}(t)$ is the input vector at time t.
- $\mathbf{w}_j(t)$ is the weight vector of neuron j at time t.
- c is the index of the winning neuron (best matching unit) for input $\mathbf{x}(t)$.
- $\alpha(t)$ is the learning rate, a monotonically decreasing function of time.
- $h_{j,c}(t)$ is the neighborhood function centered on the winning neuron c, also decreasing over time.

Neighborhood Function and Its Role The neighborhood function $h_{j,c}(t)$ typically takes a Gaussian form:

$$h_{j,c}(t) = \exp\left(-\frac{\|\mathbf{r}_j - \mathbf{r}_c\|^2}{2\sigma^2(t)}\right). \tag{9.17}$$

where:

- $\mathbf{r}_j$ and $\mathbf{r}_c$ are the positions of neurons j and c on the SOM lattice.
- $\sigma(t)$ is the neighborhood radius, which decreases over time.

This function ensures that neurons closer to the winning neuron receive larger updates, while those farther away receive smaller or zero updates. Initially, $\sigma(t)$

is large, allowing broad neighborhood cooperation, but it shrinks as training progresses, focusing updates increasingly on the winning neuron itself.

Time-Dependent Parameters Both the learning rate $\alpha(t)$ and neighborhood radius $\sigma(t)$ decrease over time, typically following exponential decay laws:

$$\alpha(t) = \alpha_0 \exp\left(-\frac{t}{\tau_\alpha}\right), \tag{9.18}$$

$$\sigma(t) = \sigma_0 \exp\left(-\frac{t}{\tau_\sigma}\right), \tag{9.19}$$

where α_0 and σ_0 are initial values, and τ_α, τ_σ are time constants controlling the decay rates.

Summary of the Six Learning Steps SOM training iteratively repeats the following six steps:

Self-Organizing Map (SOM) training pseudocode

1. **Initialize** weight vectors $\mathbf{w}_j(0)$ randomly or from samples.
2. For iteration $t = 0, \ldots, T$:
   (a) Sample an input $\mathbf{x}(t)$.
   (b) Find the best matching unit (BMU) $c = \arg\min_j \|\mathbf{x}(t) - \mathbf{w}_j(t)\|_2^2$.
   (c) Compute neighborhood coefficients $h_{j,c}(t)$.
   (d) Update every neuron:

$$\mathbf{w}_j(t+1) = \mathbf{w}_j(t) + \alpha(t)\, h_{j,c}(t)\Big(\mathbf{x}(t) - \mathbf{w}_j(t)\Big).$$

   (e) Decay learning-rate $\alpha(t)$ and neighborhood radius $\sigma(t)$ (e.g., exponentially).

Batch SOM (deterministic pass)

1. Fix centers $\mathbf{r}_j$ on the lattice; initialize $\mathbf{w}_j$ (random data or along PCA directions).
2. Repeat until convergence or max epochs:
   (a) Assign each $\mathbf{x}_n$ to its BMU c_n.
   (b) For each neuron j, update

$$\mathbf{w}_j \leftarrow \frac{\sum_n h_{j,c_n}(t)\,\mathbf{x}_n}{\sum_n h_{j,c_n}(t)}.$$

   (c) Decay $\alpha(t), \sigma(t)$.

Batch SOM (one pass per epoch) is deterministic given the assignments; it often stabilises faster than purely online updates.

These procedural steps implement three conceptual *stages*: (1) **Initialization** (seed the weight lattice), (2) **Competition** (select the best-matching unit for each input), and (3) **Cooperation** (neighborhood-weighted updates plus the associated parameter decay). Calling out these stages separately helps when comparing SOMs to other competitive-learning algorithms later in the chapter.

These steps are repeated until convergence criteria are met, such as a maximum number of iterations or a threshold on weight changes.

Stages vs. Steps It is important to distinguish between the *three stages* of SOM learning and the *six steps* described above:

- **Stages:**
  1. *Initialization:* setting up the network.
  2. *Competition:* neurons compete to respond to input.
  3. *Cooperation:* neighboring neurons cooperate to update weights.
- **Steps:** The six procedural steps in Section 9.19 that operationalize these stages during training.

9.20 Applications of Kohonen Self-Organizing Maps

Kohonen SOMs are widely used for:

- **Clustering:** Grouping similar data points without supervision.
- **Dimensionality Reduction:** Mapping high-dimensional data onto a low-dimensional space (often arranged as a discrete lattice in SOM implementations) for visualization and exploratory analysis.
- **Data Visualization:** Providing intuitive heatmaps or component planes that reveal correlations and patterns across features.

Relation to k-means and modern variants When the SOM lattice is collapsed to a single point, the algorithm reduces to a k-means-style prototype update without any notion of neighborhood (MacQueen, 1967); SOMs therefore sit between pure clustering (k-means) and manifold learning, adding a topographic prior that encourages neighboring units to represent similar inputs. Recent "neural-SOM" hybrids embed SOM-like updates inside deep networks, but still rely on the same BMU search and neighborhood-weighted updates described above.

Theory notes and recipes

Convergence/magnification: With decays $\alpha(t) \to 0$, $\sigma(t) \to 0$ and $\sum_t \alpha(t) = \infty$, $\sum_t \alpha^2(t) < \infty$, SOM updates converge under mild assumptions (Erwin et al., 1992; Cottrell and Fort, 1986). Lattice density follows data density (magnification) so dense regions attract more neurons. **Map-size/schedule recipe:** Use $5\sqrt{N}$–$10\sqrt{N}$ neurons when unsure; hex grids reduce anisotropy; toroidal lattices mitigate edges. Initialize $\mathbf{w}_j$ from data or first two PCs; $\alpha_0 \in [0.1, 0.5] \to \alpha_{\min} \approx 10^{-3}$; σ_0 equal to the map radius, decaying to 1–1.5 cells.

Related and growing variants

Neural Gas / Growing Neural Gas (Martinetz et al., 1993; Fritzke, 1994) drop the fixed lattice and learn topology or add units dynamically. **GTM** (Bishop et al.) provides a probabilistic, topographic embedding with likelihoods/uncertainty. SOMs sit between k-means (no topology) and these adaptive-topology models.

Complexity and out-of-sample mapping. A full online epoch costs $O(NM)$ (N data, M units); batch passes cost similar but fewer epochs. For large M, use approximate nearest-neighbor BMU search (k-d trees, FAISS). New points map via their BMU; optional soft responsibilities use h_{ci} for smoothing.

Theory link to other chapters. SOMs learn prototypes like the centers in Chapter 8 but add a topographic prior; quality diagnostics (QE/TE) can be tracked with the validation-curve diagnostics in Chapter 3. For task-driven embeddings, see Chapters 11 and 14; Hopfield (Chapter 10) contrasts with energy-based associative recall.

Quality measures and magnification. Two diagnostics are standard when reporting SOM quality:

- **Quantization error (QE):** average Euclidean distance between each input and its BMU. Lower QE indicates prototypes that better represent the data manifold.
- **Topographic error (TE):** fraction of inputs whose first- and second-best BMUs are not adjacent, quantifying topology preservation (magnification factor).

Tracking both metrics reveals whether the neighborhood decay is too slow (over-smoothing) or too aggressive (tearing the topology). Chapter 3's learning-curve plots suggest early-stopping heuristics: stop when QE/TE on a validation split flatten.

Exercises and lab ideas

- Train a 10×10 SOM on handwritten digits (MNIST) and plot component planes; report quantization/topographic errors as training progresses.
- Implement the six-step SOM procedure with both Gaussian and rectangular neighborhood functions and compare convergence speed.
- Visualize the effect of annealing schedules by freezing the learning rate and neighborhood radius at different epochs and observing the resulting U-Matrix.
- Compare SOM prototypes to K-means centers on the same dataset; sweep ($\sigma(t)$) schedules and map sizes; report QE/TE and a trustworthiness@k measure.

Where we head next. Chapter 10 extends the unsupervised theme by studying Hopfield networks, energy-based models that complement SOMs with associative memory dynamics and pave the way for modern attention mechanisms.

References. Full citations for works mentioned in this chapter appear in the book-wide bibliography.

10 Hopfield Networks: Introduction and Context

Learning Outcomes

- Interpret Hopfield networks as energy-minimizing recurrent systems and derive their asynchronous update rule.
- Quantify capacity, recall dynamics, and pitfalls (spurious memories, bias encodings) using simple analytical bounds.
- Relate Hopfield updates to modern energy-based models and attention mechanisms to build intuition for later chapters.

Chapter 9 focused on self-organizing maps and unsupervised feature maps; we now transition to another unsupervised/energy-based model: the *Hopfield*

network, a recurrent system that stores patterns as attractors. The roadmap in Figure 1 marks this as the energy-based branch.

Design motif

Constrain recurrence so the dynamics become a descent process: symmetric weights and an energy function turn "feedback" into "stable memory."

10.1 From Feedforward to Recurrent Neural Networks

Recall that feedforward neural networks are characterized by a unidirectional flow of information: inputs propagate through successive layers until reaching the output layer. The weights are typically updated via backpropagation, which relies on the chain rule to propagate error gradients backward through the network. Despite their success, feedforward networks do not capture the recurrent, feedback-driven dynamics observed in biological neural systems.

In contrast, *recurrent neural networks* allow cycles in their connectivity graph. This means that the state of a neuron at a given time can influence not only downstream neurons but also itself or upstream neurons through feedback loops. Such cyclic connections enable the network to maintain internal states and exhibit temporal dynamics, which are essential for tasks involving sequences and memory.

Challenges with General Recurrent Networks However, the general topology of recurrent networks introduces significant challenges:

- **Unstable dynamics:** Without careful design, recurrent networks may fail to settle into stable states, instead exhibiting chaotic or oscillatory behavior.
- **Dependence on initial conditions:** The final state of the network can be highly sensitive to the initial state, making the network's behavior unpredictable.
- **Training difficulties:** Backpropagation through time and other training methods can be computationally expensive and prone to vanishing or exploding gradients.

These issues historically limited the practical use of recurrent networks, leading to a preference for feedforward architectures in many applications.

Author's note: stabilizing recurrence

General recurrent networks can behave unpredictably because feedback can create cycles that oscillate or amplify small differences. Hopfield's key move was to restrict the architecture so the dynamics become a descent process: symmetric weights and no self-loops allow the network to be assigned an energy function that decreases under asynchronous updates. That single design choice turns "recurrent" from "chaotic" into "stable memory."

10.2 Hopfield's Breakthrough (1982)

In 1982, John Hopfield introduced a special class of recurrent networks that overcame many of these challenges by imposing specific constraints on the network architecture and weights (Hopfield, 1982). The key insights were:

- **Symmetric weights:** The connection weights between neurons are *bidirectional* and symmetric, i.e.,

$$w_{ij} = w_{ji} \quad \forall i, j, \tag{10.1}$$

  where w_{ij} is the weight from neuron j to neuron i.

- **No self-connections:** Neurons do not have self-feedback loops, so

$$w_{ii} = 0 \quad \forall i. \tag{10.2}$$

- **Binary neuron states:** Each neuron i has a state $s_i \in \{+1, -1\}$, representing *on* or *off* states, rather than continuous activations.

- **Energy-based formulation:** The network dynamics can be described by an energy function $E(\mathbf{s})$ that decreases monotonically as the network updates its states, guaranteeing convergence to a stable fixed point.

These constraints ensure that the network evolves toward local minima of the energy function, providing a natural mechanism for associative memory and pattern completion.

10.3 Network Architecture and Dynamics

Consider a Hopfield network with N neurons. The state vector is $\mathbf{s} = (s_1, s_2, \ldots, s_N)^T$, where each $s_i \in \{+1, -1\}$. The symmetric weight matrix $\mathbf{W} = [w_{ij}]$ satisfies $w_{ij} = w_{ji}$ and $w_{ii} = 0$. Throughout this discussion w_{ij} denotes the weight applied to state s_j when computing the input to neuron i, so column indices correspond to presynaptic neurons.

The *local field* or *input energy* to neuron i is defined as

$$h_i(t) = \sum_{j=1}^{N} w_{ij} s_j(t). \tag{10.3}$$

The scalar $h_i(t)$ therefore represents the total input (or *local field*) accumulated at neuron i before thresholding during iteration t.

The neuron updates its state according to the sign of $h_i(t)$ relative to a threshold θ_i:

$$s_i(t+1) = \begin{cases} +1, & h_i(t) \geq \theta_i, \\ -1, & h_i(t) < \theta_i, \end{cases} \tag{10.4}$$

Typically, thresholds θ_i are set to zero or learned as part of the model.

Interpretation: The neuron "fires" (state $+1$) if the weighted sum of inputs exceeds the threshold; otherwise, it remains "off" (state -1). This binary update rule contrasts with the continuous activation functions used in feedforward networks.

10.4 Encoding conventions

Two binary encodings are common. We primarily use $s_i \in \{-1, +1\}$ because it simplifies the energy function, but many software libraries work with $x_i \in \{0, 1\}$. Define $s = 2x - 1$ and $x = (s + 1)/2$; then

$$E_{\pm 1}(\mathbf{s}) = -\frac{1}{2} \sum_{i \neq j} w_{ij} s_i s_j + \sum_i \theta_i s_i$$

and

$$E_{01}(\mathbf{x}) = -\frac{1}{2} \sum_{i \neq j} w'_{ij} x_i x_j + \sum_i \theta'_i x_i + \text{const},$$

with $w'_{ij} = 4w_{ij}$ and $\theta'_i = 2\theta_i + 2\sum_{j\neq i} w_{ij}$ under the sign convention in $E_{\pm 1}$. The additive constant does not affect which states minimize the energy or the update dynamics. This table summarizes the correspondence:

	$\{-1,+1\}$ encoding	$\{0,1\}$ encoding
State variable	$s_i \in \{-1,+1\}$	$x_i = (s_i+1)/2$
Energy	$E_{\pm 1}(\mathbf{s})$	$E_{01}(\mathbf{x}) = E_{\pm 1}(2\mathbf{x}-\mathbf{1})$
Update rule	$s_i \leftarrow \operatorname{sign}(h_i - \theta_i)$	$x_i \leftarrow \mathbf{1}[h'_i - \theta'_i > 0]$

Whenever an equation later in the chapter uses s_i you can translate it to x_i via this affine mapping; we call out both forms only when the distinction matters. As a concrete example, the pattern $x = [1,0,1,0]$ in the $\{0,1\}$ encoding maps to $s = 2x - 1 = [+1,-1,+1,-1]$; conversely $s = [-1,+1,+1]$ corresponds to $x = [0,1,1]$.

10.5 Energy Function and Stability

Hopfield defined an energy function $E : \{-1,+1\}^N \to \mathbb{R}$ associated with the network state $\mathbf{s}$:

$$E(\mathbf{s}) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij}s_i s_j + \sum_{i=1}^{N}\theta_i s_i. \tag{10.5}$$

Because the weights are symmetric and satisfy $w_{ii} = 0$, the double sum may equivalently be written as $\sum_{i<j} w_{ij}s_i s_j$; the $\frac{1}{2}$ factor explicitly prevents counting each unordered pair twice, so removing it would scale the energy by two. Thresholds θ_i act like biases; many texts write the second term as $-\sum_i b_i s_i$ with $b_i = \theta_i$.

10.6 Hopfield Network States and Energy Function

Recall that in Hopfield networks, the state of each neuron is typically binary, either ± 1 or $0/1$. The network is characterized by symmetric weights w_{ij} between neurons and possibly thresholds θ_i. The energy function E of the network is defined to capture the "stability" of a given state vector $\mathbf{s} = (s_1, s_2, \ldots, s_N)$.

Energy function for ± 1 states: When states are bipolar, $s_i \in \{-1, +1\}$, and thresholds are zero, the energy is given by

$$E = -\frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} w_{ij} s_i s_j. \tag{10.6}$$

If thresholds θ_i are nonzero, the energy generalizes to

$$E = -\frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} w_{ij} s_i s_j + \sum_{i=1}^{N} \theta_i s_i. \tag{10.7}$$

Energy function for $\{0, 1\}$ states: When neuron states take values in $\{0, 1\}$, we denote them by x_i to avoid overloading s_i. Recenter via $s_i = 2x_i - 1$, so that $s_i \in \{-1, +1\}$. Substituting this into (10.7) yields an equivalent expression written directly in terms of the $\{0, 1\}$ variables:

$$E = -\frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} w_{ij} (2x_i - 1)(2x_j - 1) + \sum_{i=1}^{N} \theta_i (2x_i - 1). \tag{10.8}$$

Some references drop the $\frac{1}{2}$ factor when working with $\{0, 1\}$ states, but doing so merely rescales the energy because symmetry still causes every pair (i, j) to appear twice; we retain the factor to avoid double counting. The recentering view also clarifies that the dynamical behavior is the same under either encoding. One simply interprets 0 and 1 as the inactive/active states instead of -1 and $+1$.

10.7 Energy Minimization and Stable States

The fundamental goal in Hopfield networks is to find a state $\mathbf{s}$ that minimizes the energy E. Such states correspond to stable equilibria or attractors of the network dynamics.

State update dynamics: The network updates neuron states according to

$$s_i(t+1) = \text{sign}\left(\sum_{j=1}^{N} w_{ij} s_j(t) - \theta_i \right), \tag{10.9}$$

where sign(·) returns +1 for positive inputs and −1 otherwise (or applies the corresponding threshold for $\{0, 1\}$ encodings).

Asynchronous Hopfield update (pseudo-code)

1. Initialize $\mathbf{s}$ (e.g., noisy probe), set max sweeps T.
2. For $t = 1, \ldots, T$:
   (a) Pick a neuron index i (random order or cyclic sweep).
   (b) Compute $h_i = \sum_j w_{ij} s_j - \theta_i$.
   (c) Update $s_i \leftarrow \text{sign}(h_i)$.
3. Stop early if a full sweep causes no flips; else continue.

Each single-neuron update satisfies $\Delta E \leq 0$ by (10.15), so the loop converges to a local minimum of (10.11).

When neurons are updated *asynchronously* (one at a time) in any order, each step is guaranteed not to increase the energy E, ensuring convergence to a local minimum. Random or cyclic update orders satisfy the classic convergence conditions (any single flip obeys $\Delta E \leq 0$ by Equation (10.15)). Synchronous updates of all neurons, by contrast, can oscillate and are discussed in more detail in Section 10.10.

10.8 Example: Energy Calculation and State Updates

Consider a Hopfield network with three neurons, bipolar states $s_i \in \{-1, +1\}$, zero thresholds, and the symmetric weight matrix

$$W = \begin{bmatrix} 0 & 3 & -4 \\ 3 & 0 & 2 \\ -4 & 2 & 0 \end{bmatrix}.$$

Let the initial state be $\mathbf{s} = (1,\ 1,\ -1)$. Using the energy definition with the

$\frac{1}{2}$ factor to avoid double counting, we obtain

$$E(\mathbf{s}) = -\frac{1}{2}\sum_{i=1}^{3}\sum_{j=1}^{3} w_{ij}s_i s_j$$
$$= -\frac{1}{2}\Big[2\cdot 3\cdot (1)(1) + 2\cdot(-4)\cdot(1)(-1) + 2\cdot 2\cdot(1)(-1)\Big] = -5. \quad (10.10)$$

State update attempts: Flip each neuron in turn and recompute the energy:

- Flip s_1 to -1: $E(-1,1,-1) = 9$ (energy increases).
- Flip s_2 to -1: $E(1,-1,-1) = -3$ (energy increases toward zero).
- Flip s_3 to $+1$: $E(1,1,1) = -1$ (energy increases).

For clarity, Table 4 reports the energy change for each single flip relative to the current state.

Flip	New state	ΔE	Accept?
$s_1 \leftarrow -1$	(−1, 1, −1)	+14	No
$s_2 \leftarrow -1$	(1, −1, −1)	+2	No
$s_3 \leftarrow +1$	(1, 1, 1)	+4	No

Table 4: Schematic: Single-neuron flips from (1, 1, -1); all raise the energy, so the state is a local minimum.

Because every single-neuron flip raises the energy, the state $(1,1,-1)$ is a stable local minimum for this network. If the network is perturbed slightly (for instance, by flipping s_3 to $+1$ to create the noisy pattern $(1,1,1)$), asynchronous updates follow the gradient of decreasing energy and drive the system back to the stored memory $(1,1,-1)$. This illustrates how Hopfield networks perform content-addressable recall; the staircase energy trajectory in Figure 45 makes the monotone descent concrete.

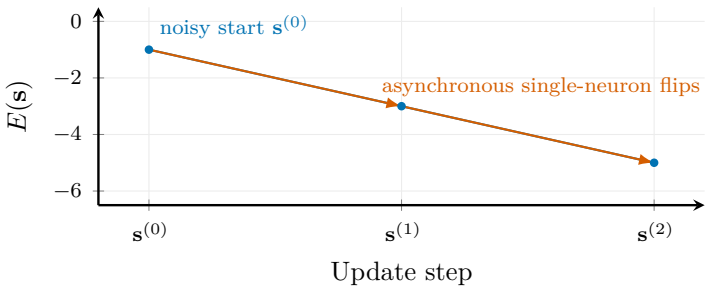

Figure 45: Schematic: Hopfield energy decreases monotonically under asynchronous updates. Starting from a noisy probe state s(0), successive single-neuron flips move downhill until the stored memory s(2) is recovered.

10.9 Energy Function and Convergence of Hopfield Networks

Recall that the Hopfield network is characterized by the energy in (10.5), repeated here for convenience:

$$E(\mathbf{s}) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij}s_i s_j + \sum_{i=1}^{N}\theta_i s_i, \tag{10.11}$$

where w_{ij} are the symmetric weights ($w_{ij} = w_{ji}$) and θ_i are the thresholds for each neuron.

Goal: Show that asynchronous updates of neuron states always decrease (or leave unchanged) the energy E, guaranteeing convergence to a local minimum.

10.9.1 Energy Change Upon Updating a Single Neuron

Consider updating neuron i from old state s_i^{old} to new state s_i^{new}. All other neuron states s_j for $j \neq i$ remain fixed. The change in energy is

$$\Delta E = E_{\text{new}} - E_{\text{old}}. \tag{10.12}$$

Using (10.11), write out the energies explicitly:

$$E_{\text{old}} = -\frac{1}{2}\sum_{k=1}^{N}\sum_{l=1}^{N} w_{kl} s_k^{\text{old}} s_l^{\text{old}} + \sum_{k=1}^{N} \theta_k s_k^{\text{old}}, \tag{10.13}$$

$$E_{\text{new}} = -\frac{1}{2}\sum_{k=1}^{N}\sum_{l=1}^{N} w_{kl} s_k^{\text{new}} s_l^{\text{new}} + \sum_{k=1}^{N} \theta_k s_k^{\text{new}}. \tag{10.14}$$

Since only s_i changes, and weights are symmetric with zero diagonal $w_{ii} = 0$, the difference simplifies to

$$\begin{aligned}
\Delta E &= E_{\text{new}} - E_{\text{old}} \\
&= -\frac{1}{2}\sum_{j=1}^{N} (w_{ij} s_i^{\text{new}} s_j + w_{ji} s_j s_i^{\text{new}}) + \theta_i s_i^{\text{new}} \\
&\quad + \frac{1}{2}\sum_{j=1}^{N} \left(w_{ij} s_i^{\text{old}} s_j + w_{ji} s_j s_i^{\text{old}} \right) - \theta_i s_i^{\text{old}} \\
&= -\sum_{j=1}^{N} w_{ij} s_j \left(s_i^{\text{new}} - s_i^{\text{old}} \right) + \theta_i \left(s_i^{\text{new}} - s_i^{\text{old}} \right) \\
&= -\left(s_i^{\text{new}} - s_i^{\text{old}} \right) \left(\sum_{j=1}^{N} w_{ij} s_j - \theta_i \right).
\end{aligned} \tag{10.15}$$

Define the *local field* h_i at neuron i as

$$h_i = \sum_{j=1}^{N} w_{ij} s_j - \theta_i. \tag{10.16}$$

Then,

$$\Delta E = -(s_i^{\text{new}} - s_i^{\text{old}}) h_i.$$

Numeric check (single flip). With two neurons, weights $w_{12} = w_{21} = 1$, thresholds $\theta_i = 0$, and current state $(s_1, s_2) = (1, -1)$, the local field at neuron 1 is $h_1 = 1 \cdot (-1) = -1$. The update rule sets $s_1^{\text{new}} = \text{sign}(h_1) = -1$, so $s_1^{\text{new}} - s_1^{\text{old}} = -2$ and $\Delta E = -(-2)(-1) = -2 < 0$, confirming the energy drop predicted by (10.15).

Modern Hopfield views and attention

Recent work (Krotov and Hopfield, 2016, 2020; Ramsauer et al., 2021) revisits Hopfield networks as dense associative memories with continuous states and softmax interactions that are closely related to Transformer attention. In this view the stored patterns play the role of keys and values, the current state or query probes the landscape, and the update rule resembles a softmax-weighted average over memories, minimizing an energy of the form $\log \sum_{\mu} \exp\big(\beta \mathbf{s}^{\top} \mathbf{m}^{\mu}\big)$ with inverse temperature β. While this book does not develop the full modern Hopfield formalism, it is helpful to remember that the energy function in (10.11) already anticipates ideas that reappear in the attention mechanisms of Chapter 13.

Continuous Hopfield / dense associative memory. Modern extensions replace the binary sign activation with a smooth, often softmax-like update that keeps neuron states continuous. Storing P real-valued patterns $\{\mathbf{m}^{\mu}\}$, the update becomes

$$\mathbf{s}^{(t+1)} = \mathrm{softmax}\Big(\beta M^{\top} \mathbf{s}^{(t)}\Big)\, M,$$

where M stacks the memory vectors and β controls sharpness. This view unifies classical Hopfield dynamics, associative memories used in few-shot meta-learning, and attention heads in Transformers: all minimize a convex energy $\log \sum_{\mu} \exp\big(\beta \mathbf{s}^{\top} \mathbf{m}^{\mu}\big)$ and pull the state toward a convex combination of stored patterns. Continuous updates are differentiable (enabling end-to-end training), offer higher storage capacity via richer nonlinearities, and connect directly to "dense associative memory" results in the modern literature.

10.9.2 Update Rule and Energy Decrease

The neuron update rule is

$$s_i^{\text{new}} = \mathrm{sign}(h_i) = \begin{cases} +1 & h_i > 0, \\ -1 & h_i < 0. \end{cases} \tag{10.17}$$

Note that $s_i^{\text{new}} \in \{-1, +1\}$, and $s_i^{\text{old}} \in \{-1, +1\}$.

Consider two cases:

- **Case 1:** $s_i^{\text{new}} = s_i^{\text{old}}$. Then $\Delta E = 0$, so the network state is unchanged.
- **Case 2:** $s_i^{\text{new}} \neq s_i^{\text{old}}$. Substituting into (10.17) gives

$$\Delta E = -2 \left(\sum_{j=1}^{N} w_{ij}\, s_j - \theta_i \right) (s_i^{\text{new}} - s_i^{\text{old}}).$$

  Because the update chooses the sign of s_i^{new} to agree with the bracketed term, $\Delta E \leq 0$, ensuring the energy never increases.

10.10 Asynchronous vs. Synchronous Updates in Hopfield Networks

Recall from the previous discussion that the Hopfield network energy function decreases monotonically with each asynchronous update of a single neuron state. This guarantees convergence to a local minimum of the energy landscape. In contrast, fully synchronous updates (flipping all neurons at once) can lead to oscillations or short cycles rather than convergence, which is why the classical convergence proofs assume asynchronous updates.

Why asynchronous updates? Suppose we attempt to update multiple neuron states simultaneously (synchronously). Consider a simple example with two neurons having states $s_1, s_2 \in \{+1, -1\}$ and weights $w_{12} = w_{21} = 10$. The energy function is:

$$E = -\frac{1}{2} \sum_{i,j} w_{ij} s_i s_j.$$

If both neurons are updated simultaneously, the energy can oscillate rather than decrease monotonically. For instance:

- Current state: $s_1 = +1, s_2 = +1$, energy $E = -20$.
- Flip both states simultaneously to $s_1 = -1, s_2 = -1$, energy $E = -20$.
- Flip back to $s_1 = +1, s_2 = +1$, energy $E = -20$.

This leads to oscillations without convergence.

Conclusion: To ensure convergence, updates must be *asynchronous* and sequential, updating one neuron at a time and respecting an update order. Revis-

iting states before all others have been updated can cause instability.

10.11 Storage Capacity of Hopfield Networks

A key question is: *How many memories can a Hopfield network reliably store and recall?*

Classical result: For a network of n neurons, the number of random patterns p that can be stored with low error is approximately

$$p \approx 0.138\, n,$$

which is a small fraction of the total number of neurons (McEliece et al., 1987). This means the storage capacity scales linearly but with a small proportionality constant; both ξ^μ and its complement $-\xi^\mu$ are fixed points, and odd mixtures of stored patterns become spurious states as p/n grows.

Inefficiency: This low capacity is why Hopfield networks are not used as practical storage devices despite their associative memory properties.

Stochastic updates (bridge to Boltzmann machines). A stochastic variant replaces the hard sign in (10.17) with probabilistic flips (e.g., Gibbs sampling). With symmetric weights this defines a Boltzmann distribution whose energy matches (10.11), linking Hopfield recall to the Boltzmann/energy-based models that underlie modern probabilistic neural networks.

10.12 Improving Storage Capacity via Weight Updates

Is it possible to improve the storage capacity by modifying the weight update rule?

Idea: Instead of fixing weights and updating states, can we update weights based on stored patterns to better encode memories?

Hebbian learning rule: Given p stored patterns $\{\mathbf{b}^1, \mathbf{b}^2, \ldots, \mathbf{b}^p\}$, each $\mathbf{b}^\mu = (b_1^\mu, b_2^\mu, \ldots, b_n^\mu)$ with $b_i^\mu \in \{+1, -1\}$, the weights are set by:

$$w_{ij} = \frac{1}{n} \sum_{\mu=1}^{p} b_i^\mu b_j^\mu, \quad w_{ii} = 0. \tag{10.18}$$

This is the classical Hebbian learning rule for Hopfield networks.

Properties:

- The diagonal terms w_{ii} are set to zero to avoid self-feedback.
- The factor $\frac{1}{n}$ normalizes the weights.
- The weights encode correlations between neuron activations across stored patterns.

Effect on capacity: Using this weight update rule, the network can store on the order of $0.138n$ random patterns reliably, which is an improvement over naive storage but still limited.

10.13 Example: Weight Calculation for a Single Pattern

Consider a fundamental memory pattern:

$$\mathbf{b} = (1, 1, 1, -1),$$

with no thresholds ($\theta_i = 0$).

Step 1: Compute outer product Form the matrix $\mathbf{B} = \mathbf{b}\mathbf{b}^\top$. Each entry $B_{ij} = b_i b_j$ captures the pairwise correlation between neurons i and j.

Step 2: Remove diagonal terms Zero the diagonal entries to obtain the weight matrix $\mathbf{W}$ with $w_{ii} = 0$. The off-diagonal values remain the same as in $\mathbf{B}$, encoding the pairwise interactions required to store the memory pattern.

10.14 Finalizing the Hopfield Network Derivation and Discussion

Recall from previous parts that the Hopfield network is a fully connected recurrent neural network designed to store and retrieve binary memory patterns $\xi^\mu \in \{-1, +1\}^N$, $\mu = 1, \ldots, P$, where N is the number of neurons and P the number of stored patterns.

The weight matrix $\mathbf{W} = [w_{ij}]$ is typically constructed using the Hebbian learning rule:

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^{P} \xi_i^\mu \xi_j^\mu, \quad w_{ii} = 0, \tag{10.19}$$

where the diagonal weights are set to zero to avoid self-feedback.

Energy Function and Convergence The network dynamics evolve asynchronously or synchronously according to the update rule:

$$s_i(t+1) = \text{sign}\left(\sum_{j=1}^{N} w_{ij} s_j(t) \right), \tag{10.20}$$

where $s_i(t) \in \{-1, +1\}$ is the state of neuron i at time t.

The Hopfield network is equipped with an energy function:

$$E(\mathbf{s}) = -\frac{1}{2} \sum_{i,j=1}^{N} w_{ij} s_i s_j, \tag{10.21}$$

which monotonically decreases (or remains constant) with each asynchronous update, guaranteeing convergence to a local minimum of E.

Memory Retrieval and Basins of Attraction The stored patterns $\{\xi^\mu\}$ correspond to local minima of the energy landscape. Starting from an initial state $\mathbf{s}(0)$ that is a noisy or partial version of a stored pattern, the network dynamics converge to the closest attractor, ideally retrieving the original memory or its complement $-\xi^\mu$.

For example, if the initial state is corrupted, the network will iteratively

update states to reduce energy until it reaches a stable point:

$$\mathbf{s}(\infty) \in \{\xi^\mu, -\xi^\mu\}.$$

Limitations: Capacity and Classification Despite its elegant memory retrieval properties, the Hopfield network has significant limitations:

- **Storage Capacity:** The maximum number of patterns $P_{\max}$ that can be reliably stored and retrieved scales approximately as $0.138N$ for large N. Beyond this, spurious minima and retrieval errors increase dramatically.
- **Spurious States:** The network may converge to spurious attractors that are not stored memories, especially when the number of stored patterns is large or when the input is heavily corrupted.
- **Classification Difficulty:** Using Hopfield networks for classification (e.g., digit recognition) is problematic. Since the network converges to the nearest energy minimum, a corrupted input pattern may converge to a wrong stored pattern or its complement. There is no guarantee that the minimum energy state corresponds to the correct class.
- **When not to use:** Large patterns and heavy loading (P/N) create a glassy energy landscape, scaling is quadratic in the number of units, and low capacity makes Hopfield networks ill-suited for high-dimensional discriminative tasks compared with the ERM models from Chapter 4 or deep models in Chapter 13.

Example: Memory Recovery Consider a Hopfield network with $N = 4$ neurons and a single stored pattern $\xi = [-1, -1, 1, -1]^T$. The weight matrix constructed via (10.18) is

$$\mathbf{W} = \frac{1}{4}\xi\xi^\top, \qquad w_{ii} = 0,$$

which numerically becomes a single symmetric matrix

$$\mathbf{W} = \frac{1}{4}\begin{pmatrix} 0 & 1 & -1 & 1 \\ 1 & 0 & -1 & 1 \\ -1 & -1 & 0 & -1 \\ 1 & 1 & -1 & 0 \end{pmatrix}.$$

The off-diagonal entries are therefore the scaled products of pattern components (e.g., $w_{12} = w_{21} = 0.25$ and $w_{13} = w_{31} = -0.25$). Thus every off-diagonal weight is simply the scaled product of the corresponding pattern entries, e.g., $w_{12} = w_{21} = 0.25$, $w_{13} = w_{31} = -0.25$, and so on.

Starting from an initial state $\mathbf{s}(0) = [-1, -1, 1, 1]^T$ (with zero thresholds $\theta_i = 0$), we apply the familiar asynchronous sign update

$$s_i \leftarrow \operatorname{sign}\Big(\sum_{j=1}^{N} w_{ij} s_j - \theta_i\Big)$$

one neuron at a time, i.e., neuron i is set to $+1$ whenever the weighted sum exceeds its threshold and to -1 otherwise. Because each update reduces the Lyapunov energy $E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij} s_i s_j + \sum_{i=1}^{N} \theta_i s_i$ from (10.11), the trajectory converges to ξ or its complement $-\xi$, demonstrating successful memory retrieval despite the initial corruption. The appearance of $-\xi$ as a fixed point is expected: the energy only depends on products $s_i s_j$, so negating all bits leaves every term unchanged.

Spurious attractors Beyond the intended memories $\{\pm\xi^\mu\}$, Hopfield networks can converge to *spurious attractors*: stable states formed by mixtures of stored patterns. These unintended minima become increasingly common as the loading factor P/N grows (here P denotes the number of stored patterns); for random patterns the practical capacity is roughly $0.138\,N$. The possibility of converging to a spurious state, or to the complemented memory rather than the original, explains why Hopfield networks are better viewed as associative memories than as discriminative classifiers.

Historical and Practical Significance The Hopfield network was revolutionary in demonstrating that artificial neural networks can model associative

memory and converge to stable states corresponding to stored memories. It laid foundational concepts for energy-based models and inspired subsequent developments in neural computation.

However, its practical use is limited by low storage capacity and sensitivity to noise. Modern networks and learning algorithms have since extended these ideas to more scalable and robust architectures.

Connections to other chapters. Hopfield networks sit between prototype-based maps and energy-based attention mechanisms. Chapter 9 and Chapter 13 provide those contexts. Their Lyapunov-style energy descent mirrors the validation-driven convergence checks from Chapter 3. Their symmetric recurrent structure also offers a counterpoint to the feedforward ERM models in Chapters 4 to 7.

Key takeaways

- Hopfield networks store binary patterns as attractors in an energy landscape defined by symmetric weights.
- Asynchronous updates monotonically reduce the Lyapunov energy, ensuring convergence to a fixed point.
- Capacity is limited and spurious attractors appear as the load P/N grows; treat Hopfield nets primarily as associative memories.

Exercises and lab ideas

- Implement a minimal example from this chapter and visualize intermediate quantities (plots or diagnostics) to match the pseudocode.
- Stress-test a key hyperparameter or design choice discussed here and report the effect on validation performance or stability.
- Re-derive one core equation or update rule by hand and check it numerically against your implementation.

Where we head next. Chapter 11 pivots from associative memory dynamics to deep feedforward architectures for perception, where convolution and pooling

build hierarchical features. The optimization discipline carries over unchanged: the ERM toolkit from Chapter 3 and the training mechanics from Chapters 6 to 7 remain central. We return to recurrence and attention in Chapter 12 and Chapter 13.

References. Full citations for works mentioned in this chapter appear in the book-wide bibliography.

11 Convolutional Neural Networks and Deep Training Tools

Learning Outcomes

After this chapter, you should be able to:

- Derive convolution/cross-correlation with stride and padding in 1D/2D.
- Explain receptive-field growth across layers and pooling effects.
- Compare loss choices for classification vs. regression and evaluation metrics.
- Connect the hinge-loss/soft-margin ideas from Chapter 3 to kernels and CNN features.
- Describe practical optimizers and regularizers (BN, dropout, weight decay).

Chapter 10 used energy to tame recurrence and create stable memories. Here we pivot back to deep feedforward models for perception, where convolutions and pooling impose a spatial inductive bias that improves sample efficiency and robustness. The roadmap in Figure 1 shows this as the deep feedforward branch.

Design motif

Keep the same statistical learning loop from Chapters 3 to 4, but move the "bias" into architecture via weight sharing and locality.

Risk & audit

- **Train/val mismatch:** augmentation, preprocessing, and normalization must be identical at evaluation time (except stochastic augmentation).
- **Resolution trade-offs:** downsampling can erase small objects; audit performance by scale and by class, not only overall accuracy.
- **BatchNorm regimes:** very small batch sizes can destabilize BN statistics; consider alternatives (layer/group norm) and audit sensitivity.
- **Shortcut learning:** CNNs can latch onto background cues; use perturbations, counterfactual crops, and slice audits.
- **Robustness:** report performance under shifts (lighting, camera, compression) and track calibration when scores are used as probabilities.

Shape reminder

Throughout this chapter we use the row-major (deep-learning) convention for batches: inputs $X \in \mathbb{R}^{B \times d_{\text{in}}}$, weights $W \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$, and biases $b \in \mathbb{R}^{d_{\text{out}}}$, with forward map $Z = XW + \mathbf{1}b^{\top}$. When we write single-example equations, you can read them as the same convention with $B = 1$. For convolution/cross-correlation we follow the standard deep-learning tensor convention (channels and spatial axes); the same shape logic applies once the tensors are flattened into matrix form.

How to read this chapter

- **Core thread (CNNs):** why locality + weight sharing → convolution/pooling mechanics → channels/feature maps → end-to-end classifiers.
- **Going deeper:** why depth works (receptive-field growth) and why CNNs displaced classical pipelines (the 2012 shift).
- **Training toolkit:** gradient-based optimization and the stabilizers you will reuse later (initialization, activations, batch norm, dropout, AdamW).

11.1 Historical Context and Motivation

The core backprop loop from earlier chapters scales to high-dimensional perception tasks, but naively applying dense multilayer perceptrons (MLPs) to images runs into two practical walls: *parameter count* and *inductive bias*. Images have strong local structure (edges, corners, textures), yet a fully connected layer treats every pixel as unrelated to its neighbors and spends parameters learning spatial patterns from scratch.

CNNs emerged as a pragmatic answer: keep end-to-end gradient training, but bake in locality and translation structure through sparse connectivity and shared weights. The result is a model class that is both more sample-efficient and more computationally viable on large grids.

11.2 Why fully connected layers break on images

Requirement for large datasets Feedforward networks typically require large amounts of labeled data to generalize well. For small datasets (e.g., Titanic survival data, movie ratings), simpler models like logistic regression or decision trees may outperform neural networks due to overfitting risks.

High-dimensional inputs and flattening Consider image data, which is naturally represented as a 2D matrix (or 3D tensor for color images). For example, a single-channel (grayscale) image of size 256×276 pixels can be represented as a matrix:

$$X \in \mathbb{R}^{256 \times 276}.$$

To input this into a traditional feedforward network, the image must be *flattened* into a vector:

$$\mathbf{x} = \text{vec}(X) \in \mathbb{R}^{70,656},$$

where $70,656 = 256 \times 276$ is the total number of pixels.

This flattening process has two major drawbacks:

- **Loss of spatial structure:** The 2D spatial relationships between pixels are ignored, which is critical for tasks like image recognition.
- **High dimensionality:** The input vector becomes very large, increasing the number of parameters and computational cost, and requiring more data to train effectively.

Implications The punchline is not that dense networks "cannot" learn images; it is that the price is too high. Without any architectural prior, you pay in parameters, data, and compute. Convolutions and pooling supply that prior: they make the model spend capacity on local patterns and reuse those patterns across the grid.

Parameter Explosion The number of weights between the input and hidden layer is:

$$70,656 \times 100 = 7,065,600, \tag{11.1}$$

and between the hidden and output layer (assuming 4 output classes) is:

$$100 \times 4 = 400. \tag{11.2}$$

Thus, the first layer alone requires learning just over 7 million parameters before we even consider deeper architectures. Coupled with the additional 400 output weights (plus biases), the optimization problem quickly becomes data-hungry and computationally expensive.

Data Requirements To reliably learn these parameters, the amount of training data must be sufficiently large. A common heuristic is that the number of training samples should be at least 10 times the number of parameters:

$$N_{\text{samples}} \geq 10 \times N_{\text{parameters}}. \tag{11.3}$$

This rule-of-thumb is intentionally conservative and should be read as guidance rather than a hard requirement; in practice, regularization, data augmentation, and strong inductive biases often permit useful models with fewer samples.

For the first layer, this implies roughly:

$$N_{\text{samples}} \gtrsim 10 \times 7,000,000 \approx 70,000,000, \tag{11.4}$$

meaning on the order of seventy million labeled images. This is an impractical requirement for most projects.

Computational and Storage Constraints Storing and processing such a large dataset requires enormous storage and computational resources. Training on hundreds of millions of images is typically infeasible for most research groups or applications without specialized infrastructure.

Overfitting Risk With millions of parameters, the model has high capacity and can easily memorize the training data, leading to overfitting. This means the network may not generalize well to unseen data, as it learns to fit noise or irrelevant details rather than meaningful features.

11.3 Sparse connectivity and parameter sharing

CNNs replace dense connections with *local receptive fields*. Each output unit connects to a small neighborhood of pixels, not the entire image. If a $k \times k$ filter scans a $H \times W$ input, the number of learned weights is k^2 (per channel), not HW. The same filter is reused at every spatial location, so a single set of parameters detects a pattern anywhere in the image. This parameter sharing is the key to scalability: it cuts the parameter count and preserves translation equivariance.

Historically, dense MLPs on image benchmarks struggled to compete with classical pipelines because the parameter count and data requirements were prohibitive. CNNs reversed that trend by tying weights and focusing on local patterns while keeping the same gradient-based training loop, so capacity grows with depth and channel count rather than with the raw pixel grid.

11.4 Convolution and pooling mechanics

For an input feature map X and filter K, the (cross-correlation) output at spatial location (i, j) is

$$(X * K)_{ij} = \sum_{u=0}^{k-1} \sum_{v=0}^{k-1} K_{uv}\, X_{i+u,\, j+v}.$$

With stride s and padding p, the output size along one axis is

$$\left\lfloor \frac{n + 2p - k}{s} \right\rfloor + 1,$$

so padding controls resolution while stride controls downsampling. Pooling then aggregates local neighborhoods (typically max or average) to build invariance and further reduce spatial size.

Convolution itself is a linear map; the nonlinearity enters when we apply an activation after each convolutional block. Stacking these linear–nonlinear stages yields hierarchical feature detectors (edges → textures → parts → objects) without abandoning the backprop training machinery.

Without padding (often called *valid* convolution), boundary pixels participate in fewer receptive fields and the spatial grid shrinks each layer. *Same* padding chooses $p = (k-1)/2$ for odd k to preserve spatial size when $s = 1$ and give edge pixels comparable influence. Larger strides reduce resolution and compute but can skip fine detail, so the padding/stride combination is a deliberate trade-off rather than a fixed rule.

11.4.1 Worked stride and padding example

Suppose an input is 6×6 and the filter is 3×3.

- **Stride $s = 1$, valid padding ($p = 0$).** Output size is $(6 - 3)/1 + 1 = 4$, so you get a 4×4 feature map.
- **Stride $s = 2$, valid padding.** Output size is $\lfloor (6 - 3)/2 \rfloor + 1 = 2$, so you get a 2×2 feature map.
- **Stride $s = 1$, same padding.** With $k = 3$, choose $p = 1$ so the output stays 6×6.

The floor in the formula is important: if $(n + 2p - k)$ is not divisible by s, the last partial window is dropped.

11.5 Pooling as nonparametric downsampling

Pooling reduces spatial size without learning new parameters: a max-pooling window keeps the strongest activation; average pooling keeps the mean. This is a nonparametric operation, so it can feel like "cheating" compared to learned filters, yet it often improves robustness by discarding small shifts and noise. Max pooling is the most common because it preserves the strongest feature response, but average and even median pooling appear in specialized settings. In modern CNNs, aggressive pooling is used sparingly; strided convolutions are a common alternative when you want learned downsampling instead of a fixed aggregation.

Author's note: pooling is a design choice

Pooling is not "more correct" than strided convolutions; it is a trade-off. If you want downsampling with learned weights, use stride. If you want a fixed local summary (often more stable early in training), max pooling is a reasonable default. Avoid padding in pooling unless you need spatial alignment with a parallel branch.

Author's note: "convolution" vs. cross-correlation

In most deep-learning libraries, the operation called "convolution" is technically *cross-correlation*: the kernel is not flipped before sliding. The name stuck because the two operations differ only by a reversal of the kernel, and the kernel is learned anyway. What matters in practice is that the "filter" is a small learned weight matrix that is applied repeatedly across space (weight sharing). Learn the shapes, the stride/padding bookkeeping, and the inductive bias; the terminology is imperfect but the idea is powerful.

11.6 Channels and feature maps

Real inputs are multi-channel. An RGB image has three channels, so a $k \times k$ filter is really $k \times k \times C_{\text{in}}$ and produces one output map. A convolutional layer applies C_{out} such filters, yielding a volume of feature maps with shape $H \times W \times C_{\text{out}}$. This is why "same" padding refers to spatial dimensions only: channel depth is set by the number of filters, not by padding.

Dimensionality bookkeeping example

Start with an input tensor of size $50 \times 50 \times 30$. Apply 10 filters of size 3×3 with stride 1 and valid padding. The spatial size becomes $50 - 3 + 1 = 48$, so the output is $48 \times 48 \times 10$. Apply 2×2 max pooling with stride 2: the spatial size becomes $\lfloor (48 - 2)/2 \rfloor + 1 = 24$, so the pooled output is $24 \times 24 \times 10$. Flattening that volume yields $24 \cdot 24 \cdot 10 = 5760$ features for a dense classifier.

11.7 Convolutional hyperparameters (what you choose up front)

- **Filter size** ($k \times k$) and **number of filters** (C_{out}).
- **Stride** s and **padding** p (valid vs. same).
- **Activation** function after each convolution.
- **Pooling** type (max/average), window size, and stride (if pooling is used).

11.8 From feature maps to classifiers

Stacks of convolutional and pooling layers produce a hierarchy of feature maps. A common design is to flatten the final maps into a vector and pass them to a small dense classifier (often a softmax layer) that predicts the class label. Backpropagation updates both the dense weights and the shared convolutional filters, so the feature extractor and classifier are learned jointly.

11.9 Multi-branch convolution blocks (Inception idea)

One practical variant is to run multiple filter sizes in parallel (for example 1×1, 3×3, and 5×5) and concatenate the resulting feature maps along the channel axis. This allows the network to capture multi-scale patterns without committing to a single kernel size. When branches must line up spatially, *same* padding is used to keep all outputs the same height and width; pooling branches often pad for this alignment even though pooling by itself is usually unpadded.

11.10 Historical Context and the 2012 Breakthrough

Before 2012, neural networks were often dismissed in many academic circles due to their poor performance on large-scale problems and the dominance of other

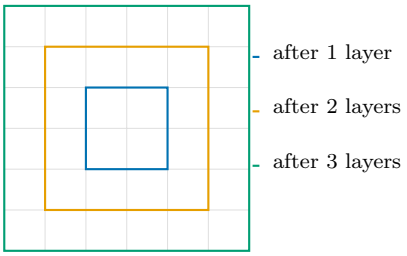

Figure 46: Schematic: Receptive field growth across depth. Even with small 3×3 kernels, stacking layers expands the spatial context seen by deeper units.

methods such as Support Vector Machines (SVMs). The sentiment was that neural networks were "fancy" but not practical or well-understood.

SVM geometry refresher. The classical soft-margin picture views an SVM as balancing a wide margin against slack variables ξ_i that widen the feasible tube so that mislabeled points incur linear penalties instead of rendering the optimization infeasible. The geometric intuition is to maximize the margin while tolerating limited violations. This highlights why SVMs were attractive when data were scarce; the hinge-loss curves in Chapter 3 supply the loss-level view of the same trade-off.

Stack depth versus receptive field. Stacking identical 3×3 filters still grows the effective receptive field: each stage wraps a thicker box around the original pixels, which explains why deep-but-narrow CNNs can capture wide spatial context without enormous kernels even when individual kernels remain small.

With the architectural motivation in place, we now focus on how these models are trained and why deep optimization remains delicate.

11.11 Training Neural Networks: Gradient-Based Optimization

Training neural networks involves minimizing a loss function $\mathcal{L}$ that measures the discrepancy between the network output and the target. The parameters (weights and biases) are updated iteratively using gradient descent or its variants.

For a weight w, the update rule is

$$w \leftarrow w - \eta \frac{\partial \mathcal{L}}{\partial w}. \tag{11.5}$$

where η is the learning rate.

Backpropagation and Gradient Computation The gradient $\frac{\partial \mathcal{L}}{\partial w}$ is computed efficiently using the backpropagation algorithm, which applies the chain rule to propagate errors backward through the network layers.

Preview: why deep optimization needs extra care As depth grows, gradient flow becomes fragile (vanishing/exploding gradients). The next sections make that failure mode explicit and summarize the mitigation toolkit used in modern CNN stacks.

11.12 Deep Network Optimization Challenges

Deep networks are difficult to optimize because the objective is highly nonconvex and the gradient signal can be distorted as it flows through many layers. We begin with the most basic pathology—vanishing and exploding gradients—and then summarize practical mitigations.

11.13 Vanishing and Exploding Gradients in Deep Networks

Recall from the previous discussion that when training deep neural networks, the backpropagation algorithm involves repeated multiplication of gradients through many layers. This repeated multiplication can cause gradients to either vanish (approach zero) or explode (grow exponentially large), leading to significant training difficulties.

Mathematical intuition Consider a deep network with L layers. Let $\delta \mathbf{W}^{(\ell)} = \nabla_{\mathbf{W}^{(\ell)}} \mathcal{L}$ denote the gradient of the loss with respect to the weights at layer ℓ. If we assume the weights are initialized identically and the derivative of the activation function is approximately constant, then the gradient at the first layer can be expressed schematically as:

$$\delta \mathbf{W}^{(1)} \approx \left(W^{(2)} D^{(2)}\right) \left(W^{(3)} D^{(3)}\right) \cdots \left(W^{(L)} D^{(L)}\right) \delta \mathbf{W}^{(L)}. \tag{11.6}$$

where W represents the weight matrix and f' is the derivative of the activation function.

Here $W^{(\ell)}$ denotes the weight matrix connecting layers $\ell - 1$ and ℓ, while $D^{(\ell)} = \operatorname{diag}\big(f'(z^{(\ell)})\big)$ collects the activation derivatives at layer ℓ. The product therefore chains together Jacobians from layers 2 through L. If the spectral norm (largest singular value) of each factor $W^{(\ell)}D^{(\ell)}$ exceeds one, then $\|\delta\mathbf{W}^{(1)}\|$ grows exponentially with L, causing **exploding gradients**. Conversely, norms less than one cause $\|\delta\mathbf{W}^{(1)}\|$ to shrink exponentially, leading to **vanishing gradients**.

Consequences

- **Exploding gradients:** The gradient values become extremely large, causing numerical instability and making the network parameters diverge during training.
- **Vanishing gradients:** The gradient values approach zero, especially in early layers, preventing those weights from updating effectively. This stalls learning in the initial layers, limiting the network's ability to learn hierarchical features.

Example: Activation function derivatives Consider the sigmoid activation function $\sigma(x) = \frac{1}{1+e^{-x}}$. Its derivative is:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)).$$

Note that $\sigma'(x)$ approaches zero when $\sigma(x)$ is near 0 or 1, i.e., when the neuron output saturates. This saturation leads to very small gradients, exacerbating the vanishing gradient problem. The derivative is maximized at $\sigma(x) = 0.5$, where $\sigma'(x) = 0.25$, so repeatedly multiplying sigmoid derivatives can shrink gradients roughly like 0.25^L across L layers.

Notation note. In this chapter $\sigma(\cdot)$ always denotes the logistic sigmoid nonlinearity, whereas symbols such as σ^2 are reserved for variances in earlier statistical chapters; context (function of an argument vs. squared scalar) distinguishes them.

11.14 Strategies to Mitigate Vanishing and Exploding Gradients

Weight initialization Initializing weights carefully can help maintain gradient magnitudes within a reasonable range. Set var $\approx 1/n$ (fan-in n).
This stabilizes signals across layers. It underlies Xavier and He.

Choice of activation function Selecting activation functions whose derivatives do not vanish easily is crucial. For example:

- **ReLU (Rectified Linear Unit):** Defined as

$$\mathrm{ReLU}(x) = \max(0, x),$$

  its derivative is 1 for positive inputs and 0 otherwise. This avoids saturation in the positive regime and helps maintain gradient flow.

- **Leaky ReLU and variants:** These allow a small, non-zero gradient when the input is negative, further mitigating dead neurons and keeping derivatives away from exact zero.

Batch normalization Batch normalization normalizes layer inputs during training, reducing the effective internal covariate shift and helping gradients maintain stable magnitudes.

Gradient clipping For exploding gradients, gradient clipping limits the maximum gradient norm during backpropagation, preventing excessively large updates.

Taken together, these tools stabilize optimization; the figures below highlight dropout, normalization, and optimizer behavior in practice.

Batch normalization. BN accelerates convergence by normalizing mini-batch statistics and learning scale/shift parameters. Figure 48 contrasts the pre- and post-normalization activation distributions; whitening the distribution keeps gradients in a well-behaved range and reduces covariate shift. In deep Transformer stacks, a closely related design choice is *pre-LN* versus *post-LN*: modern architectures typically place layer normalization *before* the residual block (pre-LN) to improve training stability on very deep networks.

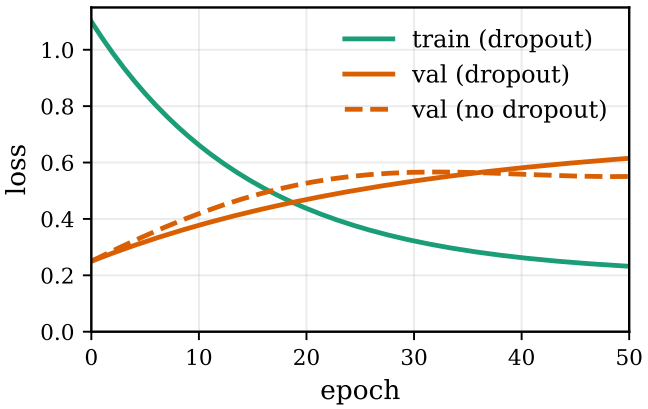

Figure 47: Schematic: Dropout effect on training/validation curves. Compared to a no-dropout baseline, validation curves flatten and generalization improves.

Adaptive optimizers. While vanilla SGD remains a workhorse, Adam and related methods adapt learning rates per-parameter (Kingma and Ba, 2015). Figure 49 summarizes the typical loss trajectories; Adam converges faster initially, whereas SGD+momentum often attains a slightly lower asymptote after fine-tuning.

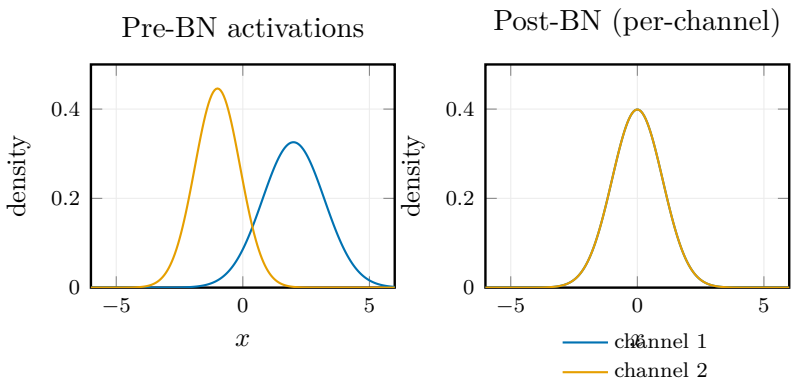

Figure 48: Schematic: Batch normalization transforms per-channel activations toward zero mean and unit variance prior to the learned affine re-scaling, stabilizing training.

Practical optimizer notes

Mixed precision. Modern CNN stacks often run activations/gradients in FP16 or BF16 while keeping master weights in FP32. Frameworks such as PyTorch AMP/TF mixed precision insert dynamic loss scaling so gradients do not underflow; the reward is higher throughput and lower memory pressure on recent GPUs/TPUs.

AdamW vs. Adam. Decoupled weight decay (AdamW) subtracts $\eta\lambda W$ outside the adaptive-moment step, avoiding the "L2-as-gradient-scaling" behavior of classical Adam and producing more predictable regularization (Loshchilov and Hutter, 2019). In code:

```
m = beta1 * m + (1-beta1) * grad
v = beta2 * v + (1-beta2) * grad**2
W -= eta * (m_hat / (sqrt(v_hat) + eps) + lambda * W)
```

Use AdamW (or SGD+momentum) when you want clean weight-decay semantics; reserve plain Adam for rapid prototyping or when adaptive steps dominate regularization.

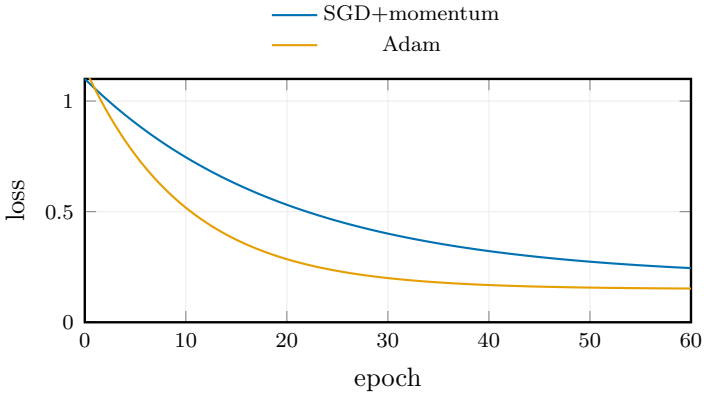

Figure 49: Schematic: Representative training curves for SGD with momentum versus Adam on the same CNN.

MLP/CNN block pseudocode (schematic)

```
function ForwardBackward(params, x, y):
  # forward
  caches = []
  a = x
  for (W, b, f) in params.layers:
    z = W @ a + b
    caches.append((a, z))
    a = f(z)
  loss, delta = params.loss(a, y)

  # backward
  grads = []
  for (W, b, f), (a_prev, z_prev) in
      reversed(list(zip(params.layers, caches))):
    grads.append((delta @ a_prev.T, delta))
    delta = (W.T @ delta) * f'(z_prev)

  params.update(grads[::-1])
  return loss
```

This omits batching, convolution strides, and optimizer detail but highlights the cache-then-backprop pattern reused throughout the deep-learning chapters.

Key takeaways

- Convolutions introduce sparse connectivity and parameter sharing, dramatically reducing parameters vs. fully connected layers.
- Padding and stride control spatial resolution; pooling aggregates features to build invariances.
- Batch normalization, dropout, and optimizer choice strongly influence training stability and generalization.
- Stacking small kernels expands the effective receptive field across depth.

Exercises and lab ideas

- Hand-compute a 1D cross-correlation with several (n, f, s, p) tuples and verify the shapes and values against a small script.
- Compare max-pool + stride 1 vs. stride-2 convolutions on a toy dataset; report accuracy and FLOPs.
- Equivariance sanity check: translate an image and confirm intermediate feature maps translate accordingly.
- Train two depth-10 CNNs on a tiny dataset: one plain, one with identity skips; compare convergence and accuracy.

Where we head next. Chapter 12 returns to sequence modeling: recurrent networks and attention mechanisms inherit many of the optimization tools discussed here, but add temporal dependencies and stateful computation.

References. Full citations for works mentioned in this chapter appear in the book-wide bibliography.

12 Introduction to Recurrent Neural Networks

Learning Outcomes

- Explain why recurrent structures are needed for sequence modeling and contrast them with feedforward nets.
- Derive the forward dynamics and backpropagation-through-time (BPTT) updates for vanilla RNN cells.
- Recognize practical stabilization techniques (gradient clipping, gating, normalization) that motivate later LSTM/Transformer chapters.

Chapter 11 showed how architectural bias (convolution/pooling) can replace some data demands while keeping the same optimization loop. This chapter turns to sequences, where the missing ingredient is memory: the model must remember enough of the past to act in the present. The roadmap in Figure 1 flags this as the sequential branch of the neural strand.

Design motif

Add recurrence, then train it by unrolling time and reusing the backprop machinery from Chapter 7.

How to read this chapter

- **Core thread (RNNs):** state and unrolling → parameter sharing → BPTT → stabilization (clipping, gating motivation) → word representations as sequence inputs.
- **Optional reminders:** brief notes on padding and autoencoders appear later; skim them if you are here primarily for RNN dynamics and training.

The statistical learning chapters (Chapters 3 to 4) established the basic training loop: choose a model class, define a loss, optimize it, and then audit generalization and calibration. The feedforward neural chapters then extended the model class from linear predictors to multilayer networks (MLPs, RBF networks, and CNNs). Those architectures have proven effective for tasks such as classification, regression, and feature extraction, but they share a common structural limitation: information flows strictly from input to output, without an internal

state that can store context over time.

12.1 Motivation for Recurrent Neural Networks

Before delving into the architecture and mathematics of RNNs, it is important to understand why feedforward networks are insufficient for certain applications. Consider the following scenario:

> *You want to predict an output at time t based not only on the input at time t, but also on inputs from previous time steps $t-1, t-2, \ldots, t-k$.*

This is a common situation in many real-world problems, such as:

- Time series forecasting (e.g., stock prices, weather data)
- Natural language processing (e.g., predicting the next word in a sentence)
- Speech recognition and synthesis
- Control systems with memory of past states

Order carries meaning even in simple settings: if it is Saturday, the next day is Sunday. In language, "out of the blue" means something sudden, whereas "the ball was blue" refers to color; the sequence makes the difference. In predictive text, "I want to buy ..." favors a different continuation than "Write a book about Teddy ..." These examples also highlight variable-length inputs: a review can be three words or three hundred. You can always build a fixed window of past inputs and feed it to a standard MLP, but that approach scales poorly as the history grows and does not share parameters across time.

Feedforward networks treat each input independently and do not have an inherent mechanism to remember or utilize past inputs. To incorporate past information, one might consider explicitly including previous inputs as part of the current input vector, but this approach quickly becomes impractical as the history length grows.

12.2 Key Idea: State and Memory in RNNs

Recurrent neural networks address this limitation by introducing a *state vector* $\mathbf{h}_t$ that summarizes information from all previous inputs up to time t. The state is updated recursively as new inputs arrive, allowing the network to maintain a form of memory.

Formally, at each time step t, the RNN receives an input vector $\mathbf{x}_t$ and updates its hidden state $\mathbf{h}_t$ according to a function f parameterized by weights θ:

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t; \theta) \tag{12.1}$$

The output $\mathbf{y}_t$ at time t is then computed as a function of the current state:

$$\mathbf{y}_t = g(\mathbf{h}_t; \theta') \tag{12.2}$$

Here, f and g are typically nonlinear functions implemented by neural network layers, and θ, θ' are learned parameters.

Interpretation: The hidden state $\mathbf{h}_t$ acts as a *summary* or *encoding* of the entire input history $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t\}$. This allows the network to make predictions that depend on the temporal context, not just the current input.

Parameter sharing across time. The same weights are reused at each time step. In the simplest formulation there are three learned matrices: W_{xh} (input to state), W_{hh} (state to state), and W_{hy} (state to output). When you unroll the recurrence, these matrices appear at every step, so the model learns a single transition rule rather than a separate set of parameters for each position in the sequence.

Unrolling makes the training graph T steps deep. Gradients from each time step accumulate onto the shared weights, and the repeated Jacobian products are precisely why long sequences revive the vanishing/exploding gradient issues from deep feedforward networks. Training this unrolled graph with standard backpropagation is known as *backpropagation through time* (BPTT).

Recurrent neural networks (RNNs) were among the first practical sequence models (Elman, 1990; Bengio et al., 1994). CNNs from Chapter 11 trade recurrence for parallel, spatially shared filters, while Chapter 13 will revisit sequence modeling without recurrence. Chapter 14 supplies the embeddings and perplexity metrics commonly paired with RNNs.

12.3 Comparison with Feedforward Networks

To contrast, a feedforward network computes the output at time t as:

$$\mathbf{y}_t = \psi(\mathbf{x}_t; \theta) \tag{12.3}$$

where ψ is a nonlinear function without any dependence on past inputs. This limits the ability of feedforward networks to model temporal dependencies unless the input vector $\mathbf{x}_t$ explicitly contains past information.

Summary: RNNs extend feedforward networks by incorporating a recurrent connection that allows information to persist across time steps, enabling modeling of sequences and temporal dynamics.

Shape reminder

We keep the row-major (deep-learning) convention: $\mathbf{x}_t \in \mathbb{R}^{d_x}$, $\mathbf{h}_t \in \mathbb{R}^{d_h}$, pre-activation $\mathbf{a}_t = \mathbf{x}_t W_{xh} + \mathbf{h}_{t-1} W_{hh} + \mathbf{b}_h$, output $\mathbf{y}_t = \mathbf{h}_t W_{hy} + \mathbf{b}_y$. Column-vector formulations simply transpose the order of factors; all stability conclusions (spectral norms of Jacobian factors) carry over.

Simple RNN at a glance

- **Objective:** Minimize cross-entropy (or another sequence loss) between targets and $p_\theta(y_t \mid h_t)$, with $h_t = f(x_t W_{xh} + h_{t-1} W_{hh} + b_h)$.
- **Key hyperparameters:** Hidden state size, BPTT truncation window, optimizer/learning rate, regularization (dropout, weight decay), gradient-clipping threshold.
- **Defaults:** tanh or ReLU activations, hidden size 128–512, Adam or SGD+momentum with clipping, layer norm or gating (LSTM/GRU) for long sequences.
- **Common pitfalls:** Vanishing/exploding gradients on long sequences, too-small hidden states, and train/test mismatch when teacher forcing is not reflected at inference.

12.4 Outline of this chapter

In this chapter, we will:

- Formally define the architecture of recurrent neural networks.
- Derive the forward and backward passes for training RNNs.
- Discuss challenges such as vanishing and exploding gradients.
- Introduce variants of RNNs designed to mitigate these challenges.
- Explore applications where RNNs provide significant advantages over feed-forward networks.

Optional reminders: padding and autoencoders (skim)

Padding (CNNs). For a 1D/2D convolution with input size n, kernel size k, stride s, and padding p, the output size is

$$\left\lfloor \frac{n + 2p - k}{s} \right\rfloor + 1.$$

When $s = 1$ and you want to preserve size, choose $p = (k - 1)/2$ for odd k.

Autoencoders. An encoder maps $\mathbf{x} \mapsto \mathbf{z}$ and a decoder maps $\mathbf{z} \mapsto \hat{\mathbf{x}}$; training uses a reconstruction loss. The idea of compressing a long input into a compact representation reappears in sequence encoders and attention models.

Vanilla RNN cell (forward + BPTT)

```
# Forward for a sequence {x_t, y_t}
h_0 = 0
for t = 1..T:
    pre_h = h_{t-1} W_hh + x_t W_xh + b_h
    h_t  = tanh(pre_h)
    yhat_t = h_t W_hy + b_y

# Backward (BPTT with optional truncation K)
delta_pre_next = 0
for t = T..1:
    delta_y = grad_loss(yhat_t, y_t)
    grad_W_hy += h_t^T delta_y
    grad_b_y += delta_y
    delta_h = delta_y W_hy^T + delta_pre_next W_hh^T
    delta_pre = delta_h .* (1 - h_t^2)
    grad_W_hh += h_{t-1}^T delta_pre
    grad_W_xh += x_t^T delta_pre
    grad_b_h += delta_pre
    delta_pre_next = delta_pre
    if t < T-K: break  # truncated BPTT
```

Use gradient clipping (e.g., clip the global norm of parameter gradients) and layer/batch normalization when sequences are long to avoid exploding/vanishing gradients.

The element-wise product in the backward step corresponds to the Hadamard factors described in the derivation in Chapter 6; we write it as `.*` to align with NumPy/Matlab notation.

Code–math dictionary. In code blocks we use ASCII identifiers such as `h_t`, `W_hh`, and `b_h`; in equations the same objects appear as $\mathbf{h}_t$, $\mathbf{W}_{hh}$, and $\mathbf{b}_h$ (boldface for vectors/matrices, subscripts for time and role).

Detailed algebraic derivations (forward/backward passes and gradient expressions) appear in Equations (12.7) to (12.8); readers are encouraged to work

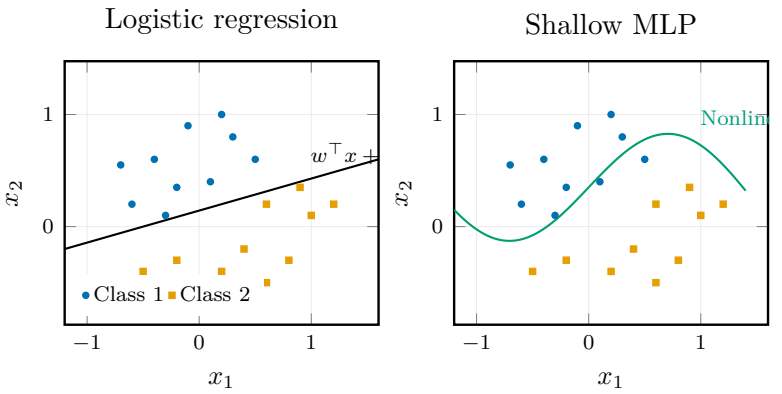

Figure 50: Schematic: Decision boundaries for logistic regression (left) versus a shallow MLP (right). Linear models carve a single hyperplane, whereas hidden units can warp the boundary to follow non-convex manifolds such as the moons dataset.

through the accompanying examples to solidify intuition.

12.5 Recap: Feedforward Building Blocks

RNNs reuse the same ingredients as multilayer perceptrons (activations, nonlinear decision boundaries, loss functions, and training heuristics) but wrap them around a temporal axis. Figure 25 from Chapter 7 highlights the canonical MLP dataflow along with common activation choices and derivatives that govern gradient flow.

Two-dimensional toy datasets remain useful for reasoning about inductive biases. Figure 50 contrasts logistic regression and a shallow MLP on the moons dataset, illustrating how additional hidden units carve nonlinear boundaries that RNN readouts later rely on when decoding the final state.

Finally, Figure 51 summarizes two diagnostics: BCE geometry and the effect of learning-rate schedules/early stopping. Here BCE (binary cross-entropy) for a binary target $y \in \{0, 1\}$ and logit z is $\mathcal{L}(z, y) = \log(1 + e^{-z})$ for y=1 and $\log(1 + e^{z})$ for y=0; the *logit* z is the pre-sigmoid score so that $\sigma(z)$ yields the predicted probability. The middle panel contrasts a conservative schedule (smooth decay) with a more aggressive one (faster initial drop but risk of oscillation), and the right panel shows early stopping triggered when validation loss ceases to improve while training loss continues decreasing. We will reuse

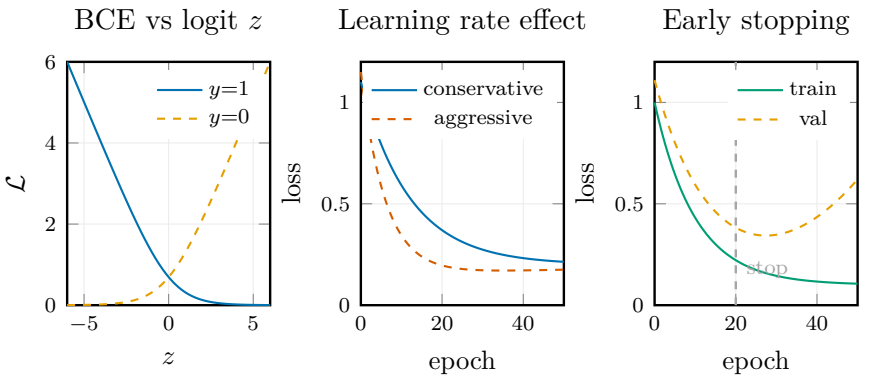

Figure 51: Schematic: Binary cross-entropy geometry (left), effect of learning-rate schedules on loss (middle), and the typical training/validation divergence that motivates early stopping (right).

these when tuning sequence models, where overfitting appears as a divergence between per-token training and validation likelihood.

Author's note: treat early stopping as the default brake

Unless there is a compelling reason to run to numerical convergence, stop as soon as the validation curve flattens while the training curve keeps dropping. Checkpoint the best weights and resume only if new data or regularization changes warrant it. That simple rule prevents most runaway experiments without elaborate hyperparameter sweeps.

LayerNorm and residual RNN tips

Layer Normalization (Ba et al., 2016) stabilizes recurrent dynamics by normalizing each hidden vector h_t across features before applying the nonlinearity; unlike BatchNorm it works with batch size 1 and handles variable-length sequences gracefully. Residual RNN stacks (adding the input of a layer back to its output) keep gradients flowing even when depth increases, mirroring the skip-connections that make deep CNNs trainable. Together, LayerNorm + residual links curb exploding/vanishing gradients and are the default when building multi-layer RNN/LSTM stacks.

Historical Note: Hopfield Networks An early influential recurrent network is the Hopfield network (Hopfield, 1982), which is a form of associative memory. Unlike modern RNNs, Hopfield networks have symmetric weights and are designed to converge to stable states representing stored patterns. While Hopfield networks are not directly used for sequence modeling, they helped establish the energy-based viewpoint that reappears in later recurrent and attention-based models.

Bidirectional extensions run two RNNs in opposite directions and concatenate their states; they are widely used in encoders for labeling tasks when the full context is available.

12.6 Input–output configurations and mathematical formulation

RNNs can map sequences to sequences in several ways:

- Many-to-one (e.g., sentiment classification): consume $x_{1:T}$, emit one label after the final state.
- One-to-many (e.g., conditional generation): condition on a context vector, then autoregressively emit a sequence.
- Many-to-many (e.g., tagging, ASR): emit y_t at every step; encoder–decoder variants compress $x_{1:T}$ then decode.
- Bidirectional encoders: run a forward and backward RNN and concatenate the states for sequence labeling or as encoder context.

Consider an input sequence $\{x_1, x_2, \ldots, x_T\}$, where each $x_t \in \mathbb{R}^d$. The RNN computes hidden states $\{h_1, h_2, \ldots, h_T\}$ and outputs $\{y_1, y_2, \ldots, y_T\}$ as follows:

$$h_0 = \mathbf{0} \quad \text{(initial hidden state)} \tag{12.4}$$

$$h_t = f(x_t W_{xh} + h_{t-1} W_{hh} + b_h), \quad t = 1, \ldots, T \tag{12.5}$$

$$y_t = g(h_t W_{hy} + b_y), \quad t = 1, \ldots, T \tag{12.6}$$

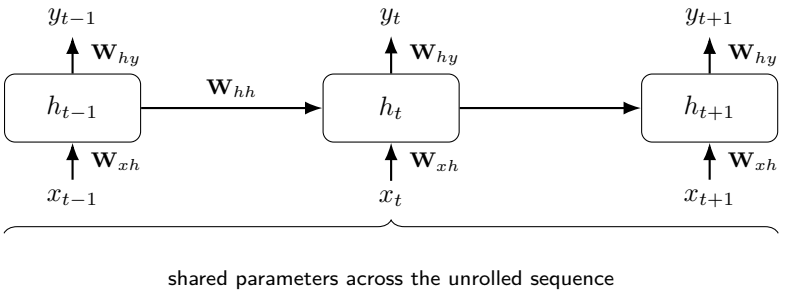

Figure 52: Schematic: Unrolling an RNN reveals repeated application of the same parameters across time steps. This view motivates backpropagation through time (BPTT), which accumulates gradients through every copy before updating the shared weights.

Shapes and masks (batch B, time T)

Inputs $X \in \mathbb{R}^{B\times T\times d_x}$; hidden states $H \in \mathbb{R}^{B\times T\times d_h}$; logits $Y \in \mathbb{R}^{B\times T\times d_o}$.
Parameters (row-major): $W_{xh} \in \mathbb{R}^{d_x\times d_h}$, $W_{hh} \in \mathbb{R}^{d_h\times d_h}$, $W_{hy} \in \mathbb{R}^{d_h\times d_o}$, biases $b_h \in \mathbb{R}^{d_h}$, $b_y \in \mathbb{R}^{d_o}$.
Padding mask $M \in \{0,1\}^{B\times T}$: loss $L = \sum_{b,t} M_{b,t}\,\mathrm{CE}(\hat{y}_{b,t}, y_{b,t}) / \sum_{b,t} M_{b,t}$.
Masks preview the padding/causal masks detailed in Chapter 13.

Unrolling and shared weights. The cleanest way to understand recurrence is to *unroll time*: the same cell is applied repeatedly, reusing the same parameters at each step. Training then becomes ordinary backpropagation on the unrolled computation graph, with gradients accumulated across every copy of the shared weights (backpropagation through time, BPTT).

12.7 Mathematical Formulation of a Simple RNN Cell

Consider a simple RNN cell with the following update equations:

$$\mathbf{h}_t = \sigma_h\left(\mathbf{x}_t\mathbf{W}_{xh} + \mathbf{h}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h\right), \tag{12.7}$$

$$\mathbf{y}_t = \sigma_y\left(\mathbf{h}_t\mathbf{W}_{hy} + \mathbf{b}_y\right). \tag{12.8}$$

12.8 Recurrent Neural Network (RNN) Architectures and Loss Computation

Recall from previous discussions that the loss function for classification tasks often involves cross-entropy terms of the form:

$$\mathcal{L} = -\sum_{i} y_i \log \hat{y}_i, \tag{12.9}$$

where y_i is the true label (often one-hot encoded) and $\hat{y}_i$ is the predicted probability for class i. When $\hat{y} = y$, the loss is zero, indicating perfect prediction.

In the context of RNNs, the total loss over a sequence is typically the sum of losses at each time step:

$$\mathcal{L}_{\text{total}} = \sum_{t=1}^{T} \mathcal{L}_t, \tag{12.10}$$

where T is the sequence length.

Forward and Backward Passes in RNNs The forward pass involves propagating inputs through the network over time steps $t = 1, \ldots, T$, producing outputs $\hat{y}_t$ at each step. After computing the loss, the backward pass computes gradients with respect to parameters by backpropagating errors through time, a process known as *Backpropagation Through Time* (BPTT).

BPTT unfolds the RNN across time steps and applies standard backpropagation through this unrolled network. The key insight is that parameters are shared across time steps, so gradients accumulate contributions from all time steps; Figure 53 highlights the simultaneous forward flow (black) and backward gradients (pink) that piggyback across every copy.

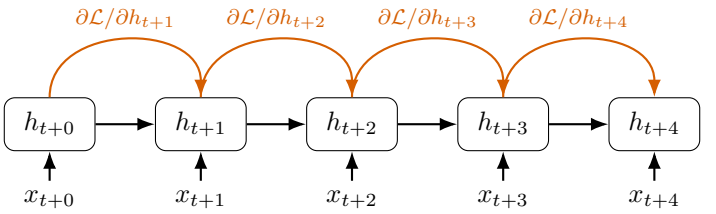

Figure 53: Schematic: Backpropagation through time (BPTT): unrolled forward pass (black) and backward gradients (pink) through time.

Truncated BPTT in practice

Unroll K steps, accumulate loss, backprop through those K steps, then detach the hidden state to stop graph growth:

```
h = h0
for t in range(T):
    h, yhat = rnn(x[t], h)
    loss += mask[t] * CE(yhat, y[t])
    if (t+1) % K == 0:
        loss.backward()
        clip_grad_norm_(model.parameters(), tau)
        opt.step(); opt.zero_grad()
        h = h.detach()  # carry state, drop graph
```

Choose K to balance memory and credit assignment (common range: 20–100 steps).

Vanishing and Exploding Gradients Because each gradient term contains products of Jacobians such as

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} = \operatorname{diag}\big(f'(\mathbf{a}_t)\big)\, \mathbf{W}_{hh}^{\top},$$

with pre-activation $\mathbf{a}_t = \mathbf{x}_t\mathbf{W}_{xh} + \mathbf{h}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h$ and elementwise nonlinearity f, long sequences multiply many such factors. Here $\mathbf{W}_{hh} \in \mathbb{R}^{d_h \times d_h}$ and

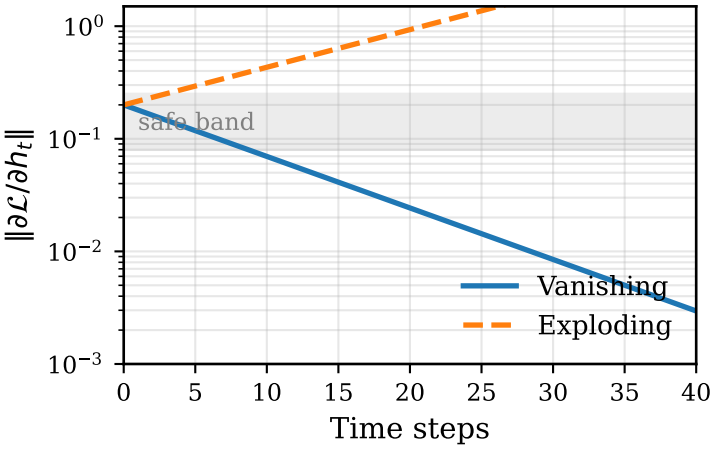

Figure 54: Schematic: Vanishing (blue) versus exploding (orange) gradients on a log scale. The gray strip highlights the stability band; the inset reminds readers that repeated Jacobian products either shrink gradients (thin blue arrows) or amplify them (thick orange arrows).

$\operatorname{diag}\big(f'(\mathbf{a}_t)\big) \in \mathbb{R}^{d_h \times d_h}$, so the Jacobian $\partial \mathbf{h}_t / \partial \mathbf{h}_{t-1}$ is a $d_h \times d_h$ matrix. If the spectral norm of each factor is below one the product decays exponentially (vanishing); norms above one cause growth (exploding). Figure 54 illustrates both behaviors across time. Practical remedies include gradient clipping, orthogonal or unitary recurrent matrices, layer normalization, and gated architectures (LSTM/GRU) that introduce additive memory paths. Empirically, vanilla RNNs often fail to preserve dependencies beyond roughly 5–10 steps in language tasks, which is why gated cells became the default for longer sequences.

Parameter Updates At each time step, the gradient of the loss with respect to parameters (e.g., weights W) depends on the chain of partial derivatives through the network states:

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial W}. \tag{12.11}$$

Because of parameter sharing, the same W influences multiple time steps, and the total gradient is the sum over these contributions.

12.9 Stabilizing Recurrent Training

Gradient clipping. A practical safeguard is to clip the global norm of the gradient when it exceeds a threshold. Figure 55 shows how clipping prevents the exploding case from destabilizing optimization while leaving the vanishing regime untouched. Orthogonal or unitary initializations for the recurrent weight matrix W_{hh} are another common trick: because orthogonal matrices preserve Euclidean norms, gradients neither explode nor vanish in the very early stages of training (before nonlinearities and data-dependent effects accumulate).

Rule of thumb: keep gradients in the safe band

Spectral norm $\|\mathbf{W}_{hh}\|_2 \approx 1$ keeps gradients roughly stable over tens of steps; clipping thresholds $\tau \in [0.5, 5]$ are sensible defaults when sequences are long. BatchNorm inside the recurrent loop is rarely helpful; prefer LayerNorm or gating to stabilize dynamics.

Author's note: do not fight vanilla RNNs

If your task needs dependencies longer than a handful of steps, do not over-tune a plain RNN. Start with clipping and a sensible truncation window, but move to GRU/LSTM once you see long-range information vanish.

Dropout in RNNs. Variational/recurrent dropout applies the same dropout mask at every time step to avoid injecting temporal noise (Gal and Ghahramani, 2016; Semeniuta et al., 2016); zoneout preserves hidden units stochastically instead of zeroing them (Krueger et al., 2017). Standard per-time-step dropout often harms sequence retention.

Teacher forcing and scheduled sampling. Sequence-to-sequence models frequently feed the ground-truth token back into the decoder during training (teacher forcing) to accelerate convergence. Figure 56 contrasts this regime with free-running inference: teacher forcing injects gold tokens at every step, whereas inference conditions the decoder on its own predictions. This mismatch is precisely what scheduled-sampling curricula aim to mitigate (Bengio et al., 2015).

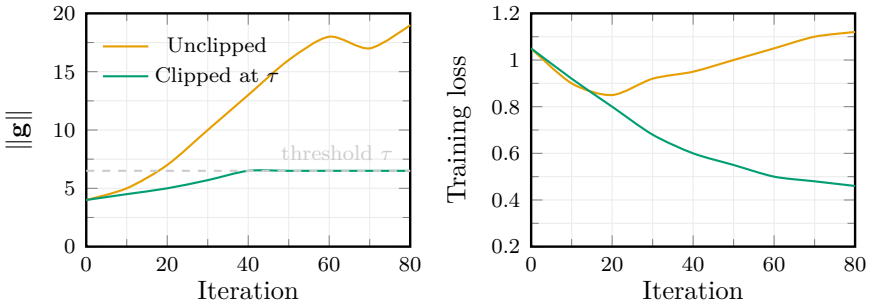

Figure 55: Schematic: Gradient norms (left) explode without clipping (orange) but remain bounded when the global norm is clipped at tau (green). Training loss (right) stabilizes as a result.

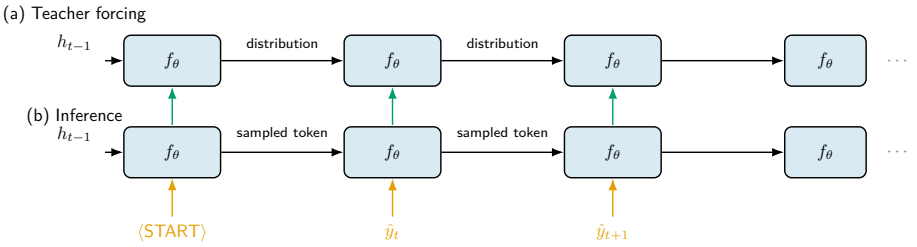

Figure 56: Schematic: Teacher forcing vs. inference in a sequence-to-sequence decoder. Gold arrows show supervised targets; orange arrows highlight autoregressive feedback that motivates scheduled sampling.

Gated cells. LSTMs (Hochreiter and Schmidhuber, 1997; Gers et al., 2000) and GRUs (Cho et al., 2014) alleviate vanishing gradients by introducing additive memory paths guarded by gates. Figures 57 and 58 present the canonical cell diagrams used later in the chapter when deriving the update equations. Intuitively, LSTM forget/input/output gates control retention, writing, and exposure of the memory c_t; GRU update/reset gates interpolate between h_{t-1} and a candidate $\tilde{h}_t$ and decide how much past context influences the candidate. In an LSTM, the state updates as $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$ and $h_t = o_t \odot \tanh(c_t)$. In a GRU, the hidden state updates as $h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$, with the reset gate r_t shaping the candidate computation. In both figures, x_t denotes the input at time t, and h_{t-1} / c_{t-1} denote the carried state(s).

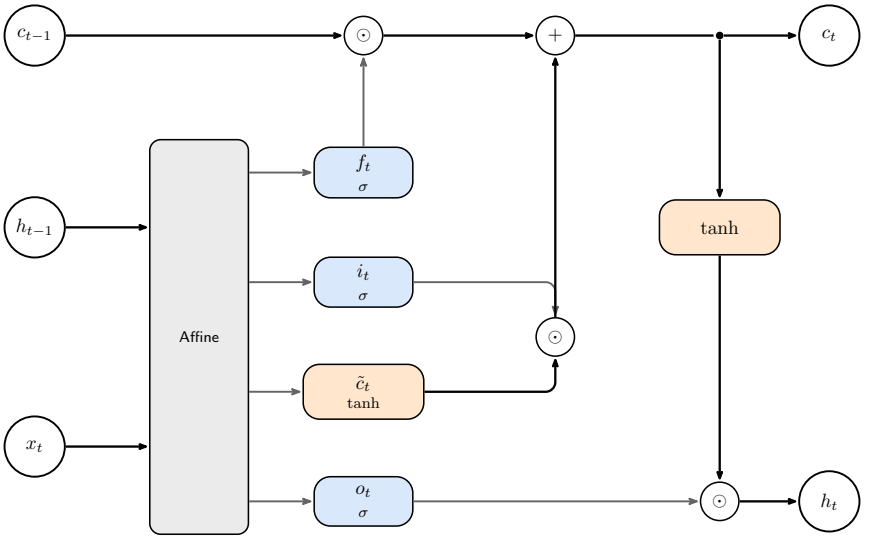

Figure 57: Schematic: Long Short-Term Memory (LSTM) cell (Hochreiter and Schmidhuber, 1997; Gers et al., 2000).

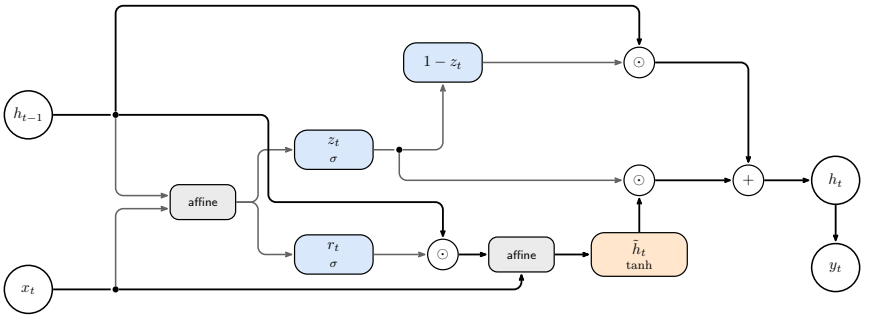

Figure 58: Schematic: Gated Recurrent Unit (GRU) cell (Cho et al., 2014).

Vanilla RNN vs. GRU vs. LSTM

Vanilla RNN: Single hidden state h_t updated via $h_t = f(x_t W_{xh} + h_{t-1} W_{hh} + b_h)$; simplest and parameter-efficient but most prone to vanishing/exploding gradients on long sequences.
GRU: Uses update and reset gates to interpolate between h_{t-1} and a candidate state; fewer parameters than LSTM, often a good default when sequences are moderately long.
LSTM: Maintains a separate cell state c_t and uses input/forget/output gates; highest parameter count but most robust on very long-range dependencies and widely used in legacy sequence models.

Minimal training loop with masks and clipping

```
h = torch.zeros(B, d_h)
for x, y, mask in loader:          # [B,T,dx], [B,T], [B,T]
    h = h.detach()                 # carry state, drop graph
    logits, h = rnn(x, h)
    loss = (mask * CE(logits, y)).sum() / mask.sum()
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), tau)
    opt.step(); opt.zero_grad()
```

Variable-length sequences are padded; the mask zeros out pads in the loss. Reset h between sequences that should not share state.

Pitfalls checklist

Mis-handled pads (loss on padded tokens); forgetting to detach hidden state across mini-batches; no clipping on long sequences; BatchNorm inside recurrence (prefer LayerNorm); teacher-forcing train/test mismatch without scheduled sampling; no masking leads to biased gradients; dropout applied independently each time step instead of variational/recurrent dropout.

Attention mechanisms. Even with gating, long sequences can challenge fixed-size hidden states. Attention augments the decoder with a content-based

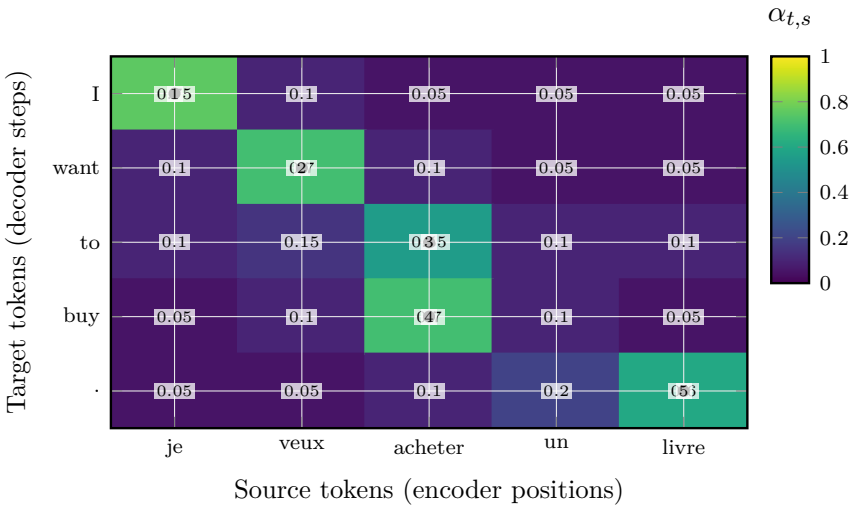

Figure 59: Schematic: Attention heatmap for a translation model. Rows are target tokens (decoder steps) and columns are source tokens (encoder positions). Each cell is an attention weight; the dot in each row marks the source position receiving the most attention.

lookup into the encoder states, as visualized in Figure 59. Bright entries correspond to encoder positions that most influence each generated token.

12.10 Representing Words for RNN Inputs

Natural language processing (NLP) requires converting words into numerical representations that RNNs can process. Since machines operate on numbers, words must be encoded appropriately.

Vocabulary Size and Word Representation Natural language has a large but finite vocabulary at any chosen granularity. Depending on whether you count words, inflections, or subword units, the effective inventory can range from tens of thousands (common token vocabularies) to far larger sets of distinct surface forms.

This finite vocabulary allows us to define a fixed-size dictionary V of words.

One-Hot Encoding A simple method to represent words is *one-hot encoding*:

- Assign each word in the vocabulary a unique index $i \in \{1, \ldots, |V|\}$.

- Represent each word as a vector $\mathbf{w} \in \mathbb{R}^{|V|}$ where all entries are zero except the i-th entry, which is 1.

For example, if $|V| = 10,000$, the word "house" might be represented as:

$$\mathbf{w}_{\text{house}} = [0, 0, \ldots, 1, \ldots, 0],$$

with the 1 in the position corresponding to "house".

This representation is sparse and high-dimensional. Conceptually the one-hot basis vectors correspond to the rows of the identity matrix $I_{|V|}$, but in practice modern models replace that fixed basis with a *learned* embedding table whose rows are trainable parameters. One convenient view is:

$$\text{one-hot words} \leftrightarrow \text{rows of } I_{|V|}, \qquad \mathbf{E} \in \mathbb{R}^{|V| \times d} \text{ trainable}, \quad \mathbf{e}_t = \mathbf{x}_t \mathbf{E},$$

so a one-hot input $\mathbf{x}_t$ simply selects the corresponding row of $\mathbf{E}$ as its dense embedding $\mathbf{e}_t$.

Why one-hot is not enough (preview). One-hot inputs are easy to implement but discard similarity: "king" and "queen" are orthogonal. The next subsection makes that failure mode concrete and motivates denser representations.

12.11 Example: Sentiment Analysis with RNNs

Consider the sentence:

"This place is great."

Each word is first converted into a numerical vector (e.g., one-hot encoded). The sequence of vectors is fed into the RNN, which processes them sequentially.

For a *many-to-one* RNN (e.g., sentiment classification), we are interested in the hidden state after processing the entire sentence. This final hidden state summarizes the contextual information and can be fed into a classifier to predict the sentiment label.

12.12 Limitations of One-Hot Encoding in Natural Language Processing

Recall that one-hot encoding represents each word in the vocabulary as a unique vector with a single 1 and zeros elsewhere. While this approach guarantees uniqueness, it fails to capture any semantic or syntactic relationships between words.

Example: Consider the sentences:

- "This place is great."
- "This place is awesome."
- "This place is good."

Using one-hot encoding, the words *great*, *awesome*, and *good* are represented as orthogonal vectors. Thus, a model trained to associate "great" with a five-star rating may not generalize to "awesome" or "good," despite their similar meanings.

Document similarity: Suppose we have two documents:

$$D_1 : \text{"I enjoyed talking to the monarchs."}$$
$$D_2 : \text{"I loved conversing with the Royals."}$$

Semantically, these sentences convey the same meaning. However, one-hot encoding treats *monarchs* and *Royals* as distinct tokens, as well as *talking* and *conversing*. Consequently, simple word-count based similarity metrics (e.g., cosine similarity on bag-of-words vectors) would yield a low similarity score, failing to capture the semantic equivalence.

Summary: One-hot encoding:

- Ignores semantic similarity between words.
- Treats synonyms and related words as completely unrelated.
- Does not capture contextual or syntactic information.

This motivates the need for richer **feature representations** of words that encode their meanings and relationships.

12.13 Feature-Based Word Representations

To encode the meaning of words, we can represent each word as a vector of *features* that capture semantic properties. These features can be handcrafted or learned, and aim to reflect qualities such as sentiment, category, or other linguistic attributes.

Example: Consider the following words:

man, woman, king, queen, orange, apple, monarch, royal

We can define features such as:

- **Gender**: male, female, neutral
- **Royalty status**: commoner, royalty
- **Age**: adult, child
- **Category**: animal, fruit, person, abstract
- **Edibility**: edible, inedible
- **Sweetness**: sweet, not sweet

Assigning numerical values to these features for each word yields a vector representation that encodes semantic information. For example:

Notes:

- The values can be binary or continuous, reflecting degrees or uncertainty (e.g., "monarch" receives a gender value of 0.5 to indicate that the term is used for multiple genders).
- High-level categories are often represented with several binary indicators (person, fruit, title, abstract) rather than a single categorical feature.
- Some features may be language- or culture-specific, and this approach requires domain knowledge and manual feature engineering.

Table 5: Schematic: Feature-based word vectorization example. Each word is mapped to a vector of graded semantic features; fractional entries (e.g., 0.5) indicate mixed usage across contexts.

Word	Gender	Royalty	Age	Person	Fruit	Title	Abstract	Sweet
man	1	0	1	1	0	0	0	0
woman	0	0	1	1	0	0	0	0
king	1	1	1	1	0	1	0	0
queen	0	1	1	1	0	1	0	0
orange	0	0	0	0	1	0	0	1
apple	0	0	0	0	1	0	0	1
monarch	0.5	1	0.5	1	0	1	0	0
royal	0	1	0.5	0	0	1	1	0

Advantages:

- Captures semantic similarity: words with similar features have similar representations.
- Enables reasoning about relationships (e.g., gender, royalty).
- Provides interpretable dimensions.

Limitations:

- Requires extensive manual effort to define and annotate features.
- May not scale well to large vocabularies or complex semantics.
- Difficult to capture contextual nuances and polysemy.

12.14 Towards Distributed Word Representations

The feature-based approach motivates the idea of **distributed representations**, where each word is represented as a dense vector in a continuous space. These vectors encode semantic and syntactic properties implicitly, often learned from large corpora.

Key idea: Instead of one-hot vectors, represent each word w as a vector $\mathbf{v}_w \in \mathbb{R}^d$, where $d \ll |V|$ (vocabulary size), such that:

$$\text{similarity}(\mathbf{v}_w, \mathbf{v}_{w'}) \approx \text{semantic similarity}(w, w')$$

Methods to obtain distributed representations Several approaches learn such embeddings automatically from corpora, including neural language models (Word2Vec CBOW and Skip-gram), matrix factorization methods (GloVe), and contextual models (ELMo, BERT). These methods leverage co-occurrence statistics to place semantically similar words nearby in the embedding space.

12.15 Semantic Relationships in Word Embeddings

We continue our exploration of word embeddings by examining how semantic relationships between words can be captured in vector space. The key insight, as demonstrated by Mikolov et al. (2013), is that certain linguistic regularities and patterns manifest as linear relationships between word vectors.

Subword tokenization and OOV handling. Modern NLP systems rarely operate on raw word types alone. To reduce vocabulary size and handle out-of-vocabulary (OOV) words, they tokenize text into *subword units*. Byte Pair Encoding (BPE) and WordPiece learn a compact inventory of frequent character sequences; words are segmented into a small number of subwords that can be recomposed by the model. FastText instead augments word vectors with character n-gram embeddings, so the representation of an unseen word is the sum of its subword vectors. Subword methods improve data efficiency, model morphology, and eliminate true OOVs while keeping sequence lengths manageable.

Example: Gender and Royalty Analogies Consider the analogy involving gender and royalty:

$$\text{king} - \text{man} + \text{woman} \approx \text{queen}.$$

This relationship suggests that the vector difference between *king* and *man* encodes the concept of "royal masculinity," and adding the vector for *woman* shifts this to "royal femininity," yielding a vector close to *queen*.

More formally, if we denote the embedding of a word w as $\mathbf{v}_w$, then the analogy can be expressed as:

$$\mathbf{v}_{\text{king}} - \mathbf{v}_{\text{man}} + \mathbf{v}_{\text{woman}} \approx \mathbf{v}_{\text{queen}}. \tag{12.12}$$

This vector arithmetic captures semantic relationships and can be used to find words that best complete analogies by maximizing cosine similarity:

$$\arg\max_{w} \cos\left(\mathbf{v}_w, \mathbf{v}_{\text{king}} - \mathbf{v}_{\text{man}} + \mathbf{v}_{\text{woman}}\right).$$

Here $\cos(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a}^\top \mathbf{b}}{\|\mathbf{a}\| \, \|\mathbf{b}\|}$ denotes cosine similarity between vectors $\mathbf{a}$ and $\mathbf{b}$.

Empirical Validation Mikolov et al. showed that these relationships hold not only for gender and royalty but also for other semantic categories such as family relations (e.g., *uncle* to *aunt*), geographical locations (e.g., *Portugal* to *Lisbon*), and cultural concepts. The distances between word vectors reflect meaningful semantic distances, such as:

$$\|\mathbf{v}_{\text{man}} - \mathbf{v}_{\text{woman}}\|_2 \approx \|\mathbf{v}_{\text{king}} - \mathbf{v}_{\text{queen}}\|_2,$$

and similarly for other pairs.

Geographical and Cultural Clustering Word embeddings also often (empirically) capture geographic and cultural proximity. For example, the embeddings for countries and their capitals frequently cluster together:

$$\mathbf{v}_{\text{Portugal}} \approx \mathbf{v}_{\text{Lisbon}}, \quad \mathbf{v}_{\text{Spain}} \approx \mathbf{v}_{\text{Madrid}}, \quad \mathbf{v}_{\text{France}} \approx \mathbf{v}_{\text{Paris}},$$

and countries that are geographically close tend to have embeddings closer in vector space (e.g., *China* is closer to *Russia* and *Japan* than to *Portugal*), although the strength of this effect depends on the corpus used for training. Throughout this chapter, statements such as $\mathbf{v}_{\text{Portugal}} \approx \mathbf{v}_{\text{Lisbon}}$ are shorthand for "the cosine similarity between the vectors exceeds a data-dependent threshold (typically > 0.8)" or, equivalently, that the two vectors lie in each other's k-nearest-neighbor list under cosine distance. These relations are empirical regularities rather than hard equalities, and the precise neighborhood structure depends on the corpus, training objective, and dimensionality of the embedding space. Figure 60 illustrates such neighborhoods after projecting embeddings to two principal components; the visual clusters make the relational analogies immediately apparent.

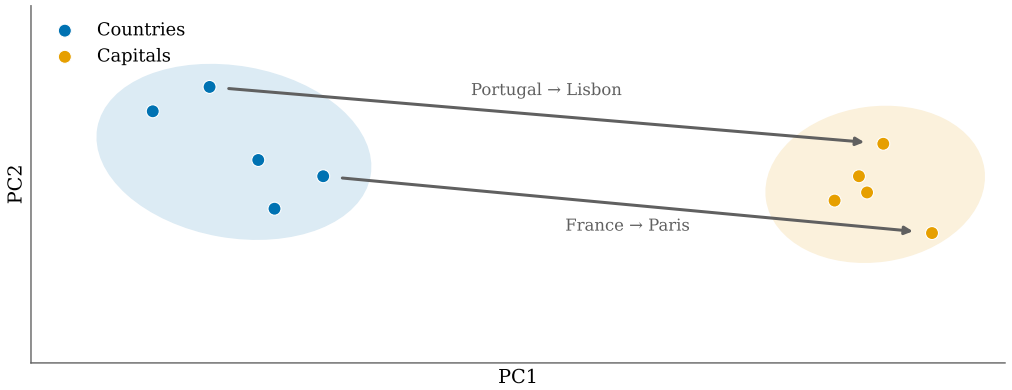

Figure 60: Schematic: Toy 2D projection of word embeddings showing neighboring clusters (countries vs. capitals). Light hulls highlight clusters; arrows show that country-to-capital displacement vectors align, a visual check on analogy structure.

12.16 Feature-Based Representation vs. One-Hot Encoding

The success of word embeddings lies in their ability to represent words as dense vectors encoding multiple latent features, as opposed to sparse one-hot vectors.

One-Hot Encoding One-hot encoding represents each word as a vector with a single 1 and zeros elsewhere. This representation is:

- **Sparse**: High-dimensional with mostly zeros (in the one-hot representation used here, the dimensionality equals the vocabulary size and only one entry is non-zero for each word).
- **Uninformative**: No notion of similarity between words.

Feature-Based Embeddings In contrast, word embeddings are dense vectors in $\mathbb{R}^d$ (typically $d = 100$ to 300) where each dimension can be interpreted as a latent feature capturing semantic or syntactic properties. These features emerge from the training process rather than being explicitly defined. The term "feature-based embedding" is non-standard in the literature; we use it here simply to stress that the coordinates behave like automatically discovered features. Most papers instead refer to these objects as *dense distributed representations*, and we always mean that same concept. Unlike the hand-crafted example below,

the latent dimensions of distributed embeddings are not usually interpretable in isolation. They capture statistical regularities uncovered automatically during training. Interpretability can sometimes be probed post hoc (e.g., via probing classifiers or dimension alignment), but there is no guarantee that any single axis corresponds cleanly to a human-understandable attribute.

Context Window Convention When we refer to the "context" of a word w_t we mean the multiset of tokens that fall within a symmetric sliding window of radius c around position t. Formally,

$$\mathcal{C}_t = \{\, w_{t-c}, \ldots, w_{t-1},\ w_{t+1}, \ldots, w_{t+c} \,\}.$$

Directional variants sometimes use only the preceding words. The co-occurrence matrix in the next section corresponds to the special case $c = 1$, where we only count the following token. Making the window definition explicit removes ambiguity about which neighboring words contribute counts to C_{ij}.

12.17 Open Questions: Feature Discovery and Representation

Two natural questions arise regarding the nature of these features:

1. **Who decides the features?** Unlike manually engineered features, the features in word embeddings are *discovered automatically* during training. There is no explicit human selection of features such as "gender" or "age." Instead, the training algorithm uncovers latent dimensions that best capture word co-occurrence statistics.

2. **How are the feature values determined?** The feature values (vector components) are learned by optimizing an objective function that encourages words appearing in similar contexts to have similar embeddings. This is typically done via self-supervised learning on large corpora. In a self-supervised setting the model creates its own supervision signal (future tokens, masked tokens, or neighboring sentences), so that no external labels are required.

Self-supervised learning of embeddings Although this learning is sometimes described informally as "unsupervised," it is more accurately *self-supervised* because the training objective uses the structure of the data itself

(e.g., predicting context words) to create targets. In self-supervised setups the model manufactures its own targets from the input (for example, masking a word and asking the network to predict it), eliminating the need for manually annotated labels.

Summary Thus, the embedding process can be viewed as a function:

$$f : \text{Vocabulary} \to \mathbb{R}^d,$$

where f is learned to encode semantic and syntactic properties implicitly, without explicit feature engineering. In matrix form we implement f by selecting the *row* of the learned embedding matrix $\mathbf{E}$ corresponding to the word of interest (row-embedding convention).

In practice we optimize objectives such as the continuous bag-of-words (CBOW) likelihood (predicts a center word from its surrounding context) and the skip-gram with negative sampling (SGNS) loss (predicts context words given a center word). These training regimes are typically optimized with SGD variants (SGD, Adam) on large corpora; Chapter 14 spells out the CBOW/skip-gram objectives in detail.

Forward pointer. We cover Word2Vec/CBOW, skip-gram, and GloVe in Chapter 14; keep the self-supervised framing in mind as we shift from RNNs to dedicated embedding objectives.

12.18 Wrapping Up the Derivations

In this chapter, we have explored the foundational concepts behind modeling sequences in natural language processing (NLP) using recurrent neural networks (RNNs). We began by considering the problem of predicting the probability of a word given its preceding context, which is central to language modeling.

Recall that the goal is to estimate the conditional probability of a word w_t given the sequence of previous words $w_1, w_2, \ldots, w_{t-1}$:

$$P(w_t \mid w_1, w_2, \ldots, w_{t-1}). \tag{12.13}$$

This probability can be modeled using an RNN, which maintains a hidden state $\mathbf{h}_t$ that summarizes the history up to time t. A common indexing choice is: consume the current token w_t (as an embedding $\mathbf{x}_t$) and predict the *next* token

w_{t+1}. This is equivalent to modeling $P(w_t \mid w_{1:t-1})$ after a one-step shift.

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t; \theta), \tag{12.14}$$

$$P(w_{t+1} \mid w_1, \ldots, w_t) = g(\mathbf{h}_t; \theta), \tag{12.15}$$

where $\mathbf{x}_t$ is the input representation (e.g., word embedding) of the word w_t, f is the recurrent update function parameterized by θ, and g maps the hidden state to a probability distribution over the vocabulary. Because the hidden state is computed recursively, $\mathbf{h}_t$ already aggregates information about the entire prefix $(w_1, \ldots, w_t)$; predicting w_{t+1} from $\mathbf{h}_t$ therefore reflects the Markovian summary that RNNs maintain. Explicitly, repeatedly substituting Equation (12.5) reveals that $\mathbf{h}_t = f(f(\cdots f(\mathbf{h}_0, \mathbf{x}_1), \ldots), \mathbf{x}_t)$, so no information is lost other than the compression inherent to the finite-dimensional state vector.

Training Objective The network is trained to maximize the likelihood of the observed sequences in a large corpus of text. Given a training sequence $(w_1, w_2, \ldots, w_T)$, the log-likelihood is:

$$\mathcal{L}(\theta) = \sum_{t=1}^{T-1} \log P(w_{t+1} \mid w_1, \ldots, w_t; \theta). \tag{12.16}$$

This objective encourages the model to assign high probability to the actual next word in the sequence, effectively learning the statistical structure of the language without explicit labeling of word relationships.

Self-supervised nature of language modeling A key insight is that no explicit labeling is required to train such models. The natural co-occurrence statistics of words in large corpora serve as implicit supervision. For example, the model learns that the word "juice" often follows "apple" because this pattern frequently appears in the training data. This is the essence of *self-supervised* learning in NLP, where the prediction targets are created directly from the input sequence.

Feature Representations The input to the RNN is typically a dense vector representation of words, known as *word embeddings*. These embeddings capture

semantic and syntactic properties of words and are learned jointly with the model parameters. The embedding matrix $\mathbf{E} \in \mathbb{R}^{V \times d}$, where V is the vocabulary size and d is the embedding dimension, maps each word index to a vector. We denote by $\mathbf{e}_{w_t} \in \{0,1\}^{1 \times V}$ the one-hot row indicator of word w_t. The embedding lookup can then be written compactly as

$$\mathbf{x}_t = \mathbf{e}_{w_t}\mathbf{E}, \tag{12.17}$$

so $\mathbf{E}[w_t]$ simply selects the row of $\mathbf{E}$ associated with w_t. The boldface $\mathbf{E}[\,\cdot\,]$ notation is intentional: it denotes array indexing into the learnable embedding matrix rather than an expectation operator $\mathbb{E}[\cdot]$. Whenever expectations appear later in the book we write them explicitly as $\mathbb{E}[\cdot]$ to avoid overload.

Summary of the Modeling Pipeline

1. Collect a large corpus of text data.
2. Tokenize the text into sequences of words.
3. Represent words as embeddings (initialized from a lookup table that is *learned jointly* with the network parameters).
4. Use an RNN to process sequences and produce hidden states.
5. Predict the next word probability distribution from the hidden state.
6. Train the model by maximizing the likelihood of the observed sequences.

LSTM and GRU equations (compact)

LSTM (single layer):

$$\begin{aligned}
\mathbf{i}_t &= \sigma(\mathbf{x}_t\mathbf{W}_i + \mathbf{h}_{t-1}\mathbf{U}_i + \mathbf{b}_i), & \mathbf{f}_t &= \sigma(\mathbf{x}_t\mathbf{W}_f + \mathbf{h}_{t-1}\mathbf{U}_f + \mathbf{b}_f),\\
\tilde{\mathbf{c}}_t &= \tanh(\mathbf{x}_t\mathbf{W}_c + \mathbf{h}_{t-1}\mathbf{U}_c + \mathbf{b}_c), & \mathbf{o}_t &= \sigma(\mathbf{x}_t\mathbf{W}_o + \mathbf{h}_{t-1}\mathbf{U}_o + \mathbf{b}_o),\\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t, & \mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t).
\end{aligned}$$

GRU:

$$\begin{aligned}
\mathbf{z}_t &= \sigma(\mathbf{x}_t\mathbf{W}_z + \mathbf{h}_{t-1}\mathbf{U}_z + \mathbf{b}_z), & \mathbf{r}_t &= \sigma(\mathbf{x}_t\mathbf{W}_r + \mathbf{h}_{t-1}\mathbf{U}_r + \mathbf{b}_r),\\
\tilde{\mathbf{h}}_t &= \tanh(\mathbf{x}_t\mathbf{W}_h + (\mathbf{r}_t \odot \mathbf{h}_{t-1})\mathbf{U}_h + \mathbf{b}_h), & \mathbf{h}_t &= (1-\mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t.
\end{aligned}$$

All gates are elementwise; σ denotes the logistic sigmoid and $\odot$ the Hadamard product.

Notation note. In the sequence-model chapters, $\sigma(\cdot)$ always denotes the logistic sigmoid gate nonlinearity; when σ is used without an argument (e.g., in earlier chapters for noise scales σ^2) it refers to a standard deviation. Context distinguishes these roles.

Key takeaways

- Language modeling is trained with self-supervision by maximizing next-token likelihood.
- Embeddings provide dense, learned word features; RNN hidden states encode context.
- Stability tools (clipping, gating, attention) enable long-range dependency modeling.

Exercises and lab ideas

- Train a many-to-one sentiment classifier; plot gradient norms with and without clipping.
- Train a small LSTM language model; compare perplexity with/without weight tying and scheduled sampling.
- Empirically sweep $\|\mathbf{W}_{hh}\|$ (spectral scaling) and reproduce vanishing/exploding behavior.
- Implement the masked, truncated-BPTT loop above and verify that pads do not affect the loss.

Where we head next. RNNs excel at streaming and moderate-context tasks but struggle with very long dependencies and parallel hardware utilization. Chapter 13 introduces attention/Transformers to address these limits; Chapter 14 revisits embeddings, perplexity, and weight tying (Press and Wolf, 2017) that pair naturally with RNN language models.

References. Full citations for works mentioned in this chapter appear in the book-wide bibliography.

13 Transformers: Attention-Based Sequence Modeling

Learning Outcomes

After this chapter, you should be able to:

- Write the scaled dot-product attention and multi-head attention formulas.
- Explain positional encodings and masking (causal/padding) in training/inference.
- Describe encoder/decoder stacks, residuals, layer norm, and training stabilizers.
- Compare RNNs vs. Transformers and know when each is preferable.
- Outline common pretraining and fine-tuning strategies (MLM/CLM, LoRA/IA3) and decoding.

Chapter 12 made the sequence problem explicit: stateful computation plus gradients that must flow across time. Transformers keep the sequence focus but remove recurrence, replacing it with attention so information can move globally in one layer. The roadmap in Figure 1 shows this as the modern sequence branch.

Design motif

Control information flow with structure (masks, normalization, residual paths) so optimization remains stable while context windows grow.

Risk & audit

- **Masking errors:** audit causal and padding masks; subtle bugs can leak future tokens or corrupt attention weights.
- **Evaluation leakage:** benchmark contamination (train/test overlap) can inflate results; prefer held-out slices and time-based splits when possible.
- **Long-context failure:** measure how quality degrades with sequence length; do not assume attention implies usable memory.
- **Calibration and confidence:** likelihood and token probabilities need not align with factual correctness; audit reliability for downstream decisions.
- **Reproducibility:** log tokenizer, data filters, and decoding settings; minor changes can swing outcomes more than architectural tweaks.

Author's note: attention allocates focus per token

It is helpful to view attention as each token asking "who else helps me understand my role," with masks and layer norms enforcing the rules of that dialogue. The scaled dot-product equations that follow merely quantify that per-token focus allocation.

13.1 Why transformers after RNNs?

Chapter 12 closed with recurrent models that process tokens sequentially, leaving them prone to vanishing/exploding gradients, limited receptive fields, and low hardware utilization. Transformers replace the recurrence with attention so every position can condition on any other in a single layer, enabling parallel hardware use and more direct long-range interactions (Vaswani et al., 2017).

13.2 Scaled Dot-Product Attention

Given query, key, value matrices $\mathbf{Q} \in \mathbb{R}^{n_q \times d_k}$, $\mathbf{K} \in \mathbb{R}^{n_k \times d_k}$, and $\mathbf{V} \in \mathbb{R}^{n_k \times d_v}$, the basic attention operation is

$$\mathrm{Attn}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}. \tag{13.1}$$

Here n_q is the sequence length of the queries and n_k the sequence length of keys/-values. We maintain the sequence-first convention from Chapter 12: rows index time positions (a "token dimension") and columns index features, while batch elements are processed independently. The $1/\sqrt{d_k}$ factor stabilizes gradients by keeping logits in a reasonable range.

Shape ledger

We treat mini-batches as $\mathbf{X} \in \mathbb{R}^{B\times n\times d_{\text{model}}}$ (batch, sequence, features). After the linear projections each head carries $\mathbf{Q}, \mathbf{K} \in \mathbb{R}^{B\times h\times n\times d_k}$, $\mathbf{V} \in \mathbb{R}^{B\times h\times n\times d_v}$, and the attention weights live in $\mathbb{R}^{B\times h\times n\times n}$. Reading dimensions in this order avoids confusion when mixing frameworks; $h \cdot d_k = d_{\text{model}}$ (often $d_v = d_k$). FFN inner widths typically 2–4$\times d_{\text{model}}$.

Complexity and memory

Naive attention is $O(n^2 d_{\text{model}})$ compute and $O(n^2)$ memory per head/layer for the attention map; this dominates long sequences. FlashAttention reduces activation I/O but keeps the quadratic arithmetic; sparse/linear variants reduce the n^2 factor by trading exactness for structure (see Longformer/BigBird/Reformer/Performer/Linformer). Causal/padding masks do not change complexity, only which entries participate.

13.3 Multi-Head Attention (MHA)

Multiple heads attend in parallel after learned linear projections:

$$\text{head}_i = \text{Attn}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V), \tag{13.2}$$

$$\text{MHA}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = [\text{head}_1; \ldots; \text{head}_h]\, \mathbf{W}^O, \tag{13.3}$$

with $\mathbf{W}_i^Q \in \mathbb{R}^{d_{\text{model}}\times d_k}$, $\mathbf{W}_i^K \in \mathbb{R}^{d_{\text{model}}\times d_k}$, $\mathbf{W}_i^V \in \mathbb{R}^{d_{\text{model}}\times d_v}$, and output projection $\mathbf{W}^O$. Figure 61 bundles scaled dot-product attention, multi-head concatenation, and the residual pre-LN block so the entire signal path is visible at a glance.

Micro attention example (2 tokens, causal mask). Let $Q = K = V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $d_k = 2$. Use a causal mask that sets all logits *above* the diagonal

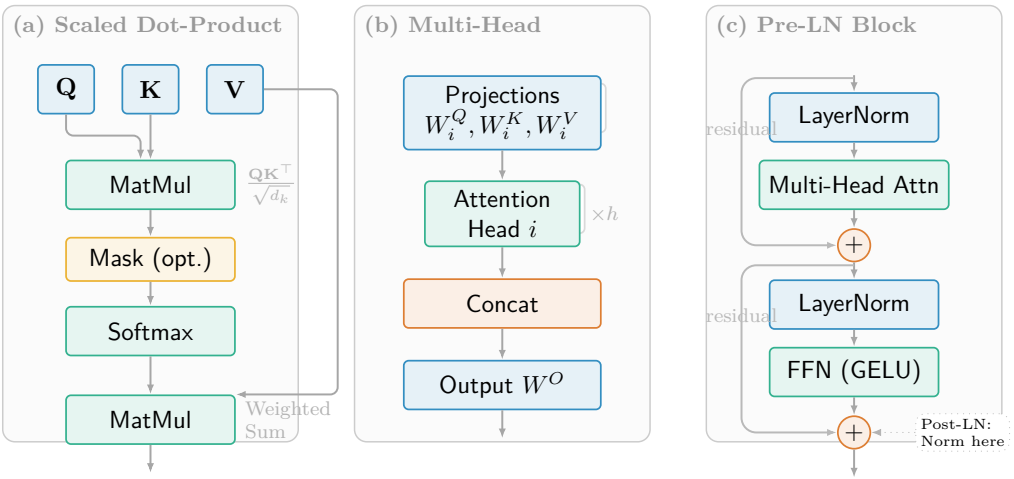

Figure 61: Schematic: Reference schematic for the Transformer. Left: scaled dot-product attention. Center: multi-head concatenation with an output projection. Right: pre-LN encoder block combining attention, FFN, and residual connections; a post-LN variant simply moves each LayerNorm after its residual add (dotted alternative, not shown).

to $-\infty$, so those entries vanish after the softmax. The unscaled score matrix is $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$; after masking and dividing by $\sqrt{2}$, the first row softmaxes to $[1, 0]$ and the second to $[\frac{e^0}{e^0+e^{1/\sqrt{2}}}, \frac{e^{1/\sqrt{2}}}{e^0+e^{1/\sqrt{2}}}]$. These are exactly the attention weights, and because V is the identity the attention output equals the weight matrix:

$$\mathrm{Attn}(Q, K, V) = \begin{bmatrix} 1 & 0 \\ \frac{1}{1+e^{1/\sqrt{2}}} & \frac{e^{1/\sqrt{2}}}{1+e^{1/\sqrt{2}}} \end{bmatrix},$$

which grounds the shapes and masking rules before we move on to larger examples. This toy case instantiates the left panel of Figure 61 with $Q = K = V = \mathbf{I}_2$ and a 2×2 causal mask.

13.4 Positional Information

Transformers lack recurrence, so order is encoded explicitly. Two common choices:

- **Sinusoidal encodings:** add $\mathbf{P}$ with fixed sine/cosine frequencies to token embeddings.

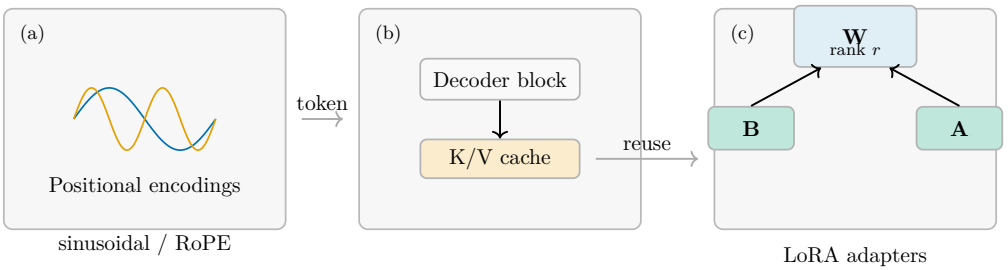

Figure 62: Schematic: Transformer micro-views. Left: positional encodings (sinusoidal/rotary) add order information. Center: KV cache stores past keys/values so decoding a new token reuses prior context. Right: LoRA inserts low-rank adapters (B A) on top of a frozen weight matrix W for parameter-efficient tuning.

- **Learned encodings:** learn a position embedding table and add to token embeddings.

Relative position encodings generalize better to long contexts and variable windows.

13.5 Masks and Training Objectives

- **Causal masks** zero out attention to future positions for autoregressive language models.
- **Padding masks** prevent attending to padding tokens in batches.
- **Pretraining:** masked language modeling (MLM; encoder) and causal LM (CLM; decoder-only). Sequence-to-sequence uses teacher forcing with encoder →decoder cross-attention.

13.6 Encoder/Decoder Stacks and Stabilizers

Each block uses residual connections and layer normalization:

$$\mathbf{H}' = \mathrm{LayerNorm}(\mathbf{H} + \mathrm{MHA}(\mathbf{H}, \mathbf{H}, \mathbf{H})), \tag{13.4}$$

$$\mathbf{H}_{\text{out}} = \mathrm{LayerNorm}(\mathbf{H}' + \mathrm{FFN}(\mathbf{H}')). \tag{13.5}$$

The feed-forward sublayer (FFN) is position-wise, typically two linear layers with a nonlinearity (e.g., GELU). Dropout and label smoothing are common.

Transformer block (pre-LN) pseudocode

```
function Block(H):
    # Pre-normalize inputs (pre-LN stabilizes deep stacks)
    H_norm = LayerNorm(H)
    attn = MHA(H_norm, H_norm, H_norm)
    H = H + Dropout(attn)
    H_norm = LayerNorm(H)
    ff = FFN(H_norm)
    return H + Dropout(ff)
```

Decoder blocks add causal masks and cross-attention with encoder states. Pre-LN (shown here) is now common because it keeps gradients well behaved for very deep stacks; post-LN (original Transformer) is still used in smaller models.

Training defaults (decoder-only, 2024)

AdamW with cosine decay and 1–3% warmup; LR $\sim 10^{-3}$ for small models, $1-2 \times 10^{-4}$ for mid-size. Weight decay ≈ 0.01 (exclude biases/LayerNorm gains). Attention/MLP dropout ≈ 0.1; clip global norm to 1.0. Mixed precision (FP16/BF16) plus gradient checkpointing for long contexts; tie input embeddings to the LM head; use causal masks for CLM, padding masks for packed batches.

One training step (decoder-only, causal mask)

```
x = tokenizer(batch_text)                  # [B, T]
mask = causal_mask(x)                      # [B, 1, T, T]
h = embed(x) + pos(x)                      # [B, T, d_model]
for block in blocks:
    h = block(h, mask)                     # pre-LN MHA + FFN
logits = lm_head(h)                        # [B, T, vocab]
loss = CE(logits[:, :-1], x[:, 1:])        # next-token
loss.backward()
clip_grad_norm_(model.parameters(), 1.0)
opt.step(); opt.zero_grad()
```

At inference, reuse cached K/V states per layer instead of recomputing attention over the full prefix.

Code–math dictionary. In code blocks, `x` is the token-index tensor (input IDs), `h` is the hidden-state array $\mathbf{H}$, `mask` is the attention mask, and `embed(x)` denotes an embedding lookup into the learned matrix $\mathbf{E}$ (written algebraically as a row-selection or $\mathbf{E}[w_t]$ in Chapter 12).

13.7 Long Contexts and Efficient Attention

Memory and compute scale quadratically with sequence length. Practical systems therefore mix several tricks:

- **Sparse or local attention** (e.g., Longformer, BigBird) to limit each query to a sliding or block-sparse neighborhood.
- **Low-rank/kernelized approximations** and recurrent chunking (Performer, Transformer-XL) so that computation/storage grows roughly linearly in context length.
- **Relative/rotary positions and cache reuse** to extrapolate beyond training lengths and avoid recomputing past keys/values during autoregressive decoding.
- **I/O-aware kernels** such as FlashAttention that stream tiles through

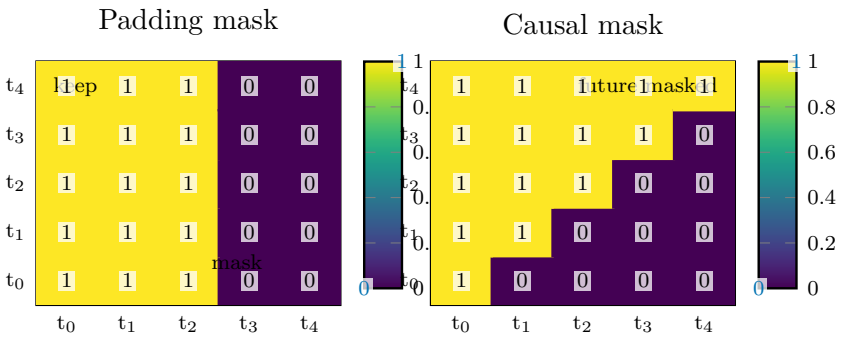

Figure 63: Schematic: Attention masks visualized as heatmaps (queries on rows, keys on columns). Left: padding mask zeroes attention into padded positions of a shorter sequence in a packed batch. Right: causal mask enforces autoregressive flow by blocking attention to future tokens.

SRAM so the quadratic pattern remains exact without exhausting memory.

13.8 Fine-Tuning and Parameter-Efficient Adaptation

Full fine-tuning updates all weights. Parameter-efficient methods (LoRA, IA3, adapters, prefix/prompt tuning) inject small trainable modules while freezing most of the base model, enabling rapid adaptation. Beyond RLHF, preference-based objectives such as DPO, KTO, or ORPO optimize alignment directly from ranked pairs without a full reinforcement-learning loop.

13.9 Decoding and Evaluation

Autoregressive generation uses greedy, beam search, top-p (nucleus), or top-k sampling. For safety and quality, monitor repetition, degeneration, and calibration. For classification tasks, prefer metrics aligned with class balance (AUPRC on imbalanced sets).

When to use which. Beam search suits short factual tasks; top-p/top-k sampling suits open-ended generation; temperature controls diversity/style. Track perplexity for language modeling (see Chapter 14) and win-rates or task metrics for downstream tasks.

13.10 Alignment (Brief)

Post-training alignment shapes model behavior to human preferences. RLHF optimizes a policy against a learned reward model; DPO offers a simpler objective based on preference pairs.

13.11 Advanced attention and efficiency notes (2024 snapshot)

- **Relative/rotary positions.** RoPE (Su et al., 2021) and ALiBi (Press et al., 2022) replace absolute sinusoidal embeddings with rotation/bias terms so extrapolating to longer sequences no longer requires re-fitting positional lookups; the trade-off is that absolute tables keep fixed anchors for classification tokens while rotary/relative schemes favour extrapolation and smoothly sliding windows.
- **KV-cache management.** Decoder-only inference stores per-layer key/-value tensors; chunked caching, paged attention, and sliding windows keep memory linear in context length. Speculative decoding and assisted decoding reuse a lightweight draft model to propose tokens that the full model verifies before committing.
- **Efficient kernels.** FlashAttention (Dao et al., 2022) computes attention blocks in streaming tiles to keep activations in SRAM. Long-context variants mix windowed attention, recurrent memory, or low-rank adapters; state-space models such as Mamba (Gu et al., 2023) provide linear-time alternatives that back-propagate through implicitly defined kernels.
- **Mixture-of-experts and routing.** Sparse MoE layers (Shazeer et al., 2017) add conditional capacity; router z-losses, capacity factors, and load-balancing losses are essential to avoid expert collapse.
- **Parameter-efficient tuning.** LoRA/QLoRA (Hu et al., 2022; Dettmers et al., 2023) insert low-rank adapters; DoRA/LoRA-FA refine the decomposition by separating direction and magnitude to better preserve pretrained weights. Adapter stacks and prefix tuning remain competitive for small target datasets.
- **Test-time scaling.** Curriculum-based sampling (nucleus, temperature annealing), classifier-free guidance, and beam-search variants all tune the

accuracy/latency frontier; plan to log decoding hyperparameters alongside checkpoints so experiments are reproducible.

13.12 RNNs vs. Transformers: When and Why

	RNN/LSTM/GRU	**Transformer**
Parallelism	Limited (sequential)	High (tokens in parallel)
Long context	Challenging (vanishing)	Natural; quadratic cost
Inductive bias	Order, recurrence	Content-based attention
Best for	Small data, streaming	Large data, global deps
Equivariance	N/A	Permutation-equivariant until positions (cf. conv translation equivariance in Chapter 11)

Practitioner box: pitfalls and checks

Pitfalls: training instability (lr too high), attention collapse, over-length inputs.
Checks: monitor loss/entropy, validation perplexity, and attention patterns on probes.
Hyperparams: heads (4–16), depth (6–24), d_{model} (256–2048), FFN multiplier ($\times 2$–$\times 4$).

Notes

Terminology: masked-LM and next-token LM are *self-supervised* (targets derived from input), not unsupervised. For embeddings downstream, we adopt row-embedding convention consistent with Chapters 7 to 8.

Key takeaways

- Attention replaces recurrence with content-based mixing, enabling highly parallel training but introducing quadratic cost in sequence length.
- Practical stability depends on details (pre-LN vs. post-LN, optimizer choices, masking, and careful decoding/evaluation).
- Architecture choices (encoder/decoder, positions, caching) are not cosmetic: they determine what the model can reuse at inference time.

Exercises and lab ideas

- Hand-compute a 2-token attention step with masking; verify against a short script.
- Implement a single-block decoder-only transformer (embed + pos + pre-LN MHA + FFN) and train on a tiny character corpus; report perplexity.
- Compare naive attention vs. FlashAttention on $n \in \{256, 1024, 4096\}$; log peak memory and tokens/sec.
- Fine-tune a base model with RoPE vs. ALiBi and evaluate extrapolation to $2\times$ the training context.
- Implement DPO on a small preference dataset; report win-rates versus the SFT baseline.

Where we head next. Chapter 14 applies these architectures to language modeling and embeddings (tokenizers, weight tying, perplexity), and covers downstream metrics (BLEU/ROUGE/AUPRC) and bias/calibration checks. Keep the encoder/decoder distinctions in mind as we map them to concrete NLP pipelines; Chapter 12 contrasts BPTT and sequential latency, and Chapter 11 notes that ViTs (Dosovitskiy et al., 2021) inherit convolutional translation equivariance only after patchification plus position encodings.

References. Full citations for works mentioned in this chapter appear in the book-wide bibliography.

14 Neural Network Applications in Natural Language Processing

Learning Outcomes

- Describe distributional semantics and the motivation for dense word embeddings.
- Derive and implement common embedding objectives (CBOW/skip-gram, negative sampling) and evaluate them via analogy tasks.
- Connect embedding quality to downstream architectures (RNNs, Transformers) and fairness considerations.

Chapter 13 showed how modern sequence models route information through attention. This chapter steps back one level to a foundational ingredient shared by RNNs and Transformers: embeddings that turn discrete tokens into geometry. The roadmap in Figure 1 places this as the application/deployment tail of the neural strand.

Design motif

Representation learning as a contract between data and objective—when you train on co-occurrence, you get both useful structure (analogies, clusters) and the biases present in the corpus.

14.1 Context and Motivation

In this chapter, we conclude our discussion on neural networks by focusing on their applications in natural language processing (NLP). Previously, we introduced the concept of representing words as inputs to a neural network, typically encoded as one-hot vectors, and obtaining as output a feature representation of these words. This feature representation captures semantic and syntactic properties of words in a continuous vector space.

A classic example illustrating the power of such representations is the analogy:

$$\text{king} - \text{man} + \text{woman} \approx \text{queen}.$$

This demonstrates that vector arithmetic on word embeddings can capture meaningful relationships between words. The goal is to find a vector space embedding where semantic similarity corresponds to geometric closeness.

14.2 Problem Statement

Given a vocabulary (corpus) of approximately 10,000 words, we want to learn a mapping from each word to a dense vector representation in a feature space of dimension d, where d is typically between 200 and 500. Formally, if the vocabulary size is V, each word w_i is initially represented as a one-hot vector $\mathbf{x}_i \in \mathbb{R}^V$, where

$$x_{ij} = \begin{cases} 1 & \text{if } j = i, \\ 0 & \text{otherwise.} \end{cases}$$

Here the row index i selects the word and the column index j specifies the position within the V-dimensional one-hot vector, so each word is associated with a unique canonical basis vector. Our objective is to learn an embedding function

$$f : \{1, \ldots, V\} \to \mathbb{R}^d,$$

such that semantic and syntactic properties of words are preserved in the embedding space.

14.3 Key Insight: Distributional Hypothesis

The foundational linguistic principle underlying word embeddings is the *distributional hypothesis*, often summarized by the phrase:

> *You shall know a word by the company it keeps.*

This idea, attributed to the linguist John Robert Firth, states that the meaning of a word can be inferred from the contexts in which it appears.

Example: The word *pretty* can have different meanings depending on context:

- In the collocation "pretty good," *pretty* functions as an adverb meaning "very" and modifies an adjective.
- In phrases such as "pretty image" or "pretty optics," *pretty* is an adjective meaning "attractive."

By explicitly examining the surrounding words (context windows of a few tokens to the left and right), we can infer the intended meaning: instances co-occurring with evaluative adjectives like "good" teach the "intensifier" sense, whereas contexts rich in nouns like "image" teach the "aesthetic" sense.

14.4 Contextual Meaning and Feature Extraction

Words appear in many different contexts, and by aggregating information from these contexts, we can infer intrinsic features of the word. For example, the contexts in which *pretty* appears with *good* or *image* help us understand its different senses.

This motivates the use of statistical models that learn word embeddings by analyzing large corpora and capturing co-occurrence patterns.

Author's note: who chooses the features?

A natural student question is: if embeddings represent "features," who decides what the features are? In modern embedding learning, the answer is: nobody writes them down explicitly. The features emerge from the training objective. By training a model to predict nearby words (or to distinguish real context pairs from random ones), the optimization process forces the hidden representation to encode whatever properties are useful for prediction. This is best viewed as *self-supervised* learning: targets come from the text itself via context windows, rather than from human labels.

14.5 Word2Vec: Two Architectures

The Word2Vec framework, introduced by Mikolov et al. (2013), operationalizes the distributional hypothesis through two main architectures:

1. **Continuous Bag of Words (CBOW):** Predicts the target word given its surrounding context words.
2. **skip-gram:** Predicts the surrounding context words given the target word.

Both architectures learn word embeddings as a byproduct of solving these prediction tasks.

14.5.1 Continuous Bag of Words (CBOW)

In CBOW, the model takes as input the context words surrounding a target word and tries to predict the target word itself. Formally, given a sequence of words $\{w_1, w_2, \ldots, w_T\}$, and a context window size n, the context for word w_t is

$$\mathcal{C}_t = \{w_{t-n}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+n}\}.$$

The CBOW model maximizes the probability

$$p(w_t \mid \mathcal{C}_t),$$

where the context words $\mathcal{C}_t$ are represented as one-hot vectors and combined (e.g., averaged) to form the input.

Example: Consider the sentence

"to buy an automatic car".

If we want to learn the embedding for the word *automatic*, the context might be $\{\text{to}, \text{buy}, \text{an}, \text{car}\}$. The CBOW model uses these context words to predict *automatic*.

14.5.2 skip-gram

Conversely, the skip-gram model takes the target word as input and tries to predict each of the context words. It maximizes

$$\prod_{w_c \in \mathcal{C}_t} p(w_c \mid w_t).$$

The product makes the modeling assumption explicit: every context word within the window contributes a likelihood factor. In practice we maximize the sum of log-probabilities $\sum_{w_c \in \mathcal{C}_t} \log p(w_c \mid w_t)$ so that each neighboring prediction provides an additive gradient signal.

This approach tends to perform better on infrequent words and captures

more detailed semantic relationships.

14.6 Mathematical Formulation of CBOW

Let the vocabulary size be V, and embedding dimension be d. Define the embedding matrix $\mathbf{W} \in \mathbb{R}^{V \times d}$, where the i-th row $\mathbf{v}_i$ is the embedding vector for word w_i. *Convention: we treat embeddings as* ***rows****; one-hot words index rows via* $\mathbf{xW}$ *(row lookup).*

14.7 Neural Network Architecture for Word Embeddings

Consider a corpus with vocabulary size $V = 10,000$ words. Our goal is to learn a dense vector representation (embedding) for each word in this vocabulary. We denote the dimensionality of the embedding space as $d = 300$.

Input Representation Each input word is represented as a one-hot row vector $\mathbf{x} \in \mathbb{R}^{1 \times V}$, where only one element is 1 (corresponding to the word index) and the rest are 0. For example, if the word "want" is the i-th word in the vocabulary, then $\mathbf{x}_i = 1$ and $\mathbf{x}_j = 0$ for $j \neq i$.

Network Structure We consider a simple feedforward neural network with:

- An input layer of size V (one-hot encoded words).
- A hidden layer of size $d = 300$, which will serve as the embedding layer.
- An output layer of size V, which predicts the target word.

The weight matrix between the input and hidden layer is denoted as

$$W \in \mathbb{R}^{V \times d}.$$

Each row $W_{i,:}$ corresponds to the embedding vector of the i-th word.

Forward Pass Given an input word represented by $\mathbf{x}$, the hidden layer output $\mathbf{h} \in \mathbb{R}^d$ is computed as:

$$\mathbf{h} = \mathbf{x}W, \tag{14.1}$$

where $\mathbf{x}$ is a $1 \times V$ vector and W is $V \times d$, resulting in $\mathbf{h}$ of size $1 \times d$.

Because $\mathbf{x}$ is one-hot, this operation simply selects the row of W corresponding to the input word, i.e., the embedding vector for that word.

Output Layer The hidden layer output $\mathbf{h}$ is then multiplied by an output matrix $W_{\text{out}} \in \mathbb{R}^{d \times V}$ to produce the output logits $\mathbf{z} \in \mathbb{R}^V$:

$$\mathbf{z} = \mathbf{h} W_{\text{out}}. \tag{14.2}$$

These logits are then passed through a softmax function to produce a probability distribution over the vocabulary:

$$\hat{\mathbf{y}}_j = \frac{\exp(z_j)}{\sum_{k=1}^{V} \exp(z_k)}, \quad j = 1, \ldots, V. \tag{14.3}$$

Training Objective The target output $\mathbf{y}$ is also a one-hot vector corresponding to the word we want to predict (e.g., the word "automatic"). The training objective is to minimize the cross-entropy loss between the predicted distribution $\hat{\mathbf{y}}$ and the target $\mathbf{y}$:

$$\mathcal{L} = -\sum_{j=1}^{V} y_j \log \hat{y}_j. \tag{14.4}$$

Backpropagation and Weight Updates During training, the weights $\mathbf{W}$ and $\mathbf{W}_{\text{out}}$ are updated via backpropagation to minimize $\mathcal{L}$. This process adjusts the embeddings in $\mathbf{W}$ so that words appearing in similar contexts have similar vector representations.

14.8 Context window and sequential input

Suppose we use a context window of size 4 words surrounding the target word. For example, to predict the word “automatic” in the phrase “to buy an automatic car,” the context words are

$$\text{to}, \quad \text{buy}, \quad \text{an}, \quad \text{car}.$$

Each context word is represented as a one-hot vector and fed into the network. Each one-hot vector shares the same embedding matrix $\mathbf{W}$; multiplying $\mathbf{xW}$ is

an efficient row lookup because $\mathbf{x}$ is one-hot.

Input Sequence Processing The same embedding lookup is applied to each context token. If $\mathbf{x}_{t+j}$ is the one-hot vector for the word at position $t+j$, then its embedding is

$$\mathbf{h}_{t+j} = \mathbf{x}_{t+j}\mathbf{W}.$$

The hidden representations $\mathbf{h}^{(i)}$ for each context word can be combined (e.g., concatenated or averaged) before passing to the output layer to predict the target word.

Dimensionality and Sparsity Note that the input vectors $\mathbf{x}_{t+j}$ are extremely sparse (one-hot), and the embedding matrix $\mathbf{W}$ is large (10,000 $\times$ 300, for example). However, the multiplication $\mathbf{x}_{t+j}\mathbf{W}$ is efficient because it selects a single row of $\mathbf{W}$ per input word.

14.9 Interpretation of the Weight Matrix W

The matrix $\mathbf{W}$ can be interpreted as a lookup table: the i-th row is the embedding for word w_i, and $\mathbf{xW}$ (with one-hot $\mathbf{x}$) selects that row directly.

14.10 Word Embeddings: Continuous Bag of Words (CBOW) and skip-gram models

Recall from the previous discussion that word embeddings are dense vector representations of words learned from large corpora, capturing semantic and syntactic properties. Two foundational models for learning such embeddings are the Continuous Bag of Words (CBOW) and skip-gram models, both introduced in the Word2Vec framework.

14.10.1 Continuous Bag of Words (CBOW)

In CBOW, the objective is to predict a target word given its surrounding context words. Formally, given a sequence of words $w_1, w_2, \ldots, w_T$, and a context window of size c, the model predicts the word w_t based on the context words $\{w_{t-c}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+c}\}$.

The input to the model is a one-hot encoded vector representing the context words. Since each word is represented as a one-hot vector of dimension V (the vocabulary size), the input is a sparse vector with a single 1 and zeros elsewhere.

The embedding matrix $\mathbf{W} \in \mathbb{R}^{V \times d}$ maps each word to a d-dimensional dense vector (embedding).

The CBOW model computes the average of the embeddings of the context words using an *input* embedding matrix $\mathbf{W}$ and predicts with a separate *output* embedding matrix $\mathbf{W}_{\text{out}}$:

$$\mathbf{h} = \frac{1}{2c} \sum_{\substack{-c \le j \le c \\ j \neq 0}} \mathbf{x}_{t+j} \mathbf{W} \tag{14.5}$$

where $\mathbf{x}_{t+j}$ is the one-hot vector for the context word at position $t + j$, and c denotes the *half-window* size (there are $2c$ context words around w_t when the document is long enough).

This hidden representation $\mathbf{h}$ is then used to predict the target word w_t via a softmax layer:

$$P(w_t \mid \text{context}) = \frac{\exp(\mathbf{h}\, \mathbf{u}_{w_t})}{\sum_{w=1}^{V} \exp(\mathbf{h}\, \mathbf{u}_w)} \tag{14.6}$$

where $\mathbf{u}_w$ is the output vector corresponding to word w. It is useful to think of the set of output vectors as the *columns* of a second matrix $\mathbf{W}_{\text{out}} \in \mathbb{R}^{d \times V}$; although $\mathbf{W}_{\text{out}}$ often starts as a copy of $\mathbf{W}^\top$, the two sets of embeddings are optimized independently during training. Many modern implementations optionally *tie* these matrices so that $\mathbf{W}_{\text{out}} = \mathbf{W}^\top$, reducing parameters and encouraging symmetry between input and output spaces.

Training proceeds by maximizing the log-likelihood over the corpus, adjusting the embedding matrix $\mathbf{W}$ and output vectors $\mathbf{u}_w$ to improve prediction accuracy. After sufficient training, the rows of $\mathbf{W}$ serve as the learned word embeddings.

Key Insight: Because the input vectors are one-hot encoded, the multiplication $\mathbf{x}_{t+j}\mathbf{W}$ simply selects the *row* of $\mathbf{W}$ corresponding to the context word w_{t+j}. This makes the embedding matrix $\mathbf{W}$ a lookup table of word features.

14.10.2 skip-gram model

The skip-gram model reverses the CBOW objective: it uses the current word to predict its surrounding context words. Given a center word w_t, the model aims to maximize the probability of each context word w_{t+j} within a window c:

$$\prod_{\substack{-c \le j \le c \\ j \neq 0}} P(w_{t+j} \mid w_t) \tag{14.7}$$

The input is the one-hot vector $\mathbf{x}_t$ representing the center word, which is projected into the embedding space via the same input embedding matrix $\mathbf{W} \in \mathbb{R}^{V \times d}$:

$$\mathbf{h} = \mathbf{x}_t \mathbf{W} \tag{14.8}$$

Each context word w_{t+j} is predicted by applying a softmax over the output vectors (again using the output-embedding matrix $\mathbf{W}_{\text{out}}$):

$$P(w_{t+j} \mid w_t) = \frac{\exp\big(\mathbf{h}\,\mathbf{u}_{w_{t+j}}\big)}{\sum_{w=1}^{V} \exp(\mathbf{h}\,\mathbf{u}_w)} \tag{14.9}$$

where $\mathbf{u}_w$ are the output vectors as before.

Training Objective: Maximize the log-likelihood of the context words given the center word over the entire corpus. Compare to the RNN language-model objective in Chapter 12: both predict nearby tokens, but here the context is a fixed sliding window rather than a learned recurrent state.

Interpretation: The skip-gram model learns embeddings such that words appearing in similar contexts have similar vector representations.

14.10.3 Computational Challenges: Softmax Normalization

Both CBOW and skip-gram models require computing the softmax normalization over the entire vocabulary V, which can be very large (e.g., $V = 10,000$

or more). The denominator in equations (14.6) and (14.9) involves summing exponentials over all vocabulary words:

$$Z = \sum_{w=1}^{V} \exp(\mathbf{h}\,\mathbf{u}_w) \tag{14.10}$$

Recipe: skip-gram with negative sampling

Preprocess: tokenize (BPE/WordPiece or whitespace), lowercase if appropriate, drop rare words below a cutoff, subsample frequent words with $t \approx 10^{-5}$.
Hyperparameters: window $c = 2$–5 (often dynamic/symmetric), embedding dim $d = 100$–300, negatives $k = 5$–20, unigram noise $P_n(w) \propto f(w)^{0.75}$, LR on the order of 10^{-3}–10^{-2}.
Per-positive loss (one context word):
$-\log \sigma(\mathbf{h}\,\mathbf{u}_{\text{pos}}) - \sum_{i=1}^{k} \log \sigma(-\mathbf{h}\,\mathbf{u}_{\text{neg},i})$; complexity $O(k)$ vs. $O(V)$ for full softmax.

This is computationally expensive, especially when training on large corpora.

Approximate Solutions: To address this, several approximation techniques have been proposed:

- **Hierarchical softmax:** factor the softmax into a tree so each update touches only a $\log V$ path.
- **Negative sampling:** replace the full softmax with k binary logistic losses against sampled "noise" words.

14.11 Efficient Training of Word Embeddings: Hierarchical Softmax and Negative Sampling

Recall from the previous discussion that computing the full softmax over a large vocabulary is computationally expensive. Specifically, given an input word, calculating the probability distribution over all possible output words in the vocabulary requires a normalization over potentially millions of terms, which is prohibitive in practice.

There are two primary strategies to address this computational bottleneck:

1. Hierarchical Softmax Hierarchical softmax replaces the flat softmax layer with a binary tree representation of the vocabulary. Each word corresponds to a leaf node, and the probability of a word is decomposed into the probabilities of traversing the path from the root to that leaf. This reduces the computational complexity from $O(V)$ to $O(\log V)$, where V is the vocabulary size.

The key idea is to organize words so that frequent words have shorter paths, thus further improving efficiency. During training, only the nodes along the path to the target word are updated, avoiding the need to compute scores for all words.

2. Negative Sampling Negative sampling is an alternative approximation that simplifies the objective by transforming the multi-class classification problem into multiple binary classification problems.

- For each observed word-context pair (w, c), the model aims to distinguish the true pair from randomly sampled *negative* pairs (w, c'), where c' is drawn from a noise distribution.
- Instead of computing probabilities over the entire vocabulary, the model only updates parameters for the positive pair and a small number of negative samples.

Example: Consider the sentence:

"I want to buy a big brick house in the city."

Suppose the context word is `brick`. The true target word is `house`. Negative samples might be `lion`, `bake`, or `big` (although `big` appears in the sentence, it can still be sampled as a negative example depending on the sampling strategy). Negative draws occasionally colliding with real context words is harmless. The associated losses simply push the model to separate the sampled pair unless the data provide strong evidence to the contrary.

Training Objective with Negative Sampling Define the logistic regression classifier that, given an input word vector $\mathbf{v}_w$ and an output word vector $\mathbf{v}'_c$,

predicts whether the pair (w, c) is observed (label 1) or a negative sample (label 0).

The probability that the pair is observed is modeled as:

$$p(D = 1 \mid w, c) = \sigma(\mathbf{v}_w \, \mathbf{v}'_c) \tag{14.11}$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function.

The training objective for one positive pair (w, c) and k negative samples $\{c'_1, \ldots, c'_k\}$ is:

$$\log \sigma(\mathbf{v}_w \, \mathbf{v}'_c) + \sum_{i=1}^{k} \log \sigma\big(- \mathbf{v}_w \, \mathbf{v}'_{c'_i}\big) \tag{14.12}$$

where each c'_i is drawn independently from the noise distribution $P_n(c)$. A widely used practical choice is $P_n(w) \propto f(w)^{0.75}$, where $f(w)$ is the empirical unigram frequency; this slightly downweights extremely frequent words while still sampling them often enough to learn robust embeddings.

Tiny worked example (skip-gram with $k = 2$). Suppose the center word is `brick` with embedding $\mathbf{v}_{\text{brick}}$, the true context is `house` ($\mathbf{v}'_{\text{house}}$), and we sample two negatives `lion`, `bake`. We compute:

$$L = \log \sigma(\mathbf{v}_{\text{brick}} \, \mathbf{v}'_{\text{house}}) + \log \sigma(-\mathbf{v}_{\text{brick}} \, \mathbf{v}'_{\text{lion}}) + \log \sigma(-\mathbf{v}_{\text{brick}} \, \mathbf{v}'_{\text{bake}}).$$

Gradients push $\mathbf{v}_{\text{brick}}$ closer to $\mathbf{v}'_{\text{house}}$ (if the dot product is too small) and simultaneously push it away from $\mathbf{v}'_{\text{lion}}$ and $\mathbf{v}'_{\text{bake}}$. Only these three context vectors update this step, so the cost stays $O(k)$ regardless of vocabulary size.

Interpretation: The model learns to assign high similarity scores to true word-context pairs and low similarity scores to randomly sampled pairs, effectively learning meaningful embeddings without computing the full softmax. The expectation over the noise distribution is estimated by the empirical average across the k sampled negatives in (14.12). Unlike noise-contrastive estimation (NCE), negative sampling is not a consistent estimator of the normalized softmax probabilities; it is best viewed as a task-specific approximation that yields high-quality embeddings rather than calibrated class posteriors.

Backpropagation: The gradients are computed only for the positive pair and the sampled negative pairs, drastically reducing computation.

Connection to PMI (Levy & Goldberg). A useful theoretical lens relates skip-gram with negative sampling (SGNS) to pointwise mutual information (PMI). Under common choices of windowing and negative sampling distribution, SGNS implicitly factorizes a *shifted* PMI matrix such that inner products approximate:

$$\mathbf{v}_i \, \mathbf{u}_k \;\approx\; \mathrm{PMI}(i,k) - \log k, \quad \text{where} \quad \mathrm{PMI}(i,k) = \log \frac{P(i,k)}{P(i)P(k)}.$$

This connection helps explain why SGNS and GloVe often yield similar geometric regularities despite different training objectives: both methods recover statistics of co-occurrence up to monotone transformations and weighting.

14.12 Local Context vs. Global Matrix Factorization Approaches

Word embedding methods can be broadly categorized into two classes based on how they utilize context information:

1. Local Context Window Methods These methods focus on the immediate context of a word within a fixed-size window. Examples include:

- Continuous Bag-of-Words (CBOW)
- skip-gram

They learn embeddings by predicting a word given its neighbors (CBOW) or predicting neighbors given a word (skip-gram). These methods are computationally efficient and capture syntactic and semantic relationships based on local co-occurrence patterns.

2. Global Matrix Factorization Methods These methods consider the entire corpus to build a global co-occurrence matrix X, where each entry X_{ij} counts how often word i co-occurs with word j across the corpus.

- Latent Semantic Analysis (LSA) is an early example, which applies singular value decomposition (SVD) to the co-occurrence matrix.

- More recent methods include GloVe (Global Vectors), which factorizes a weighted log-count matrix (Pennington et al., 2014).

Example: Co-occurrence Matrix Suppose the vocabulary size is V. The co-occurrence matrix $X \in \mathbb{R}^{V \times V}$ is defined as:

$$X_{ij} = \text{number of times word } i \text{ appears in the context of word } j$$

This matrix is *sparse* and *large* (especially when V runs into the hundreds of thousands), so storing it explicitly or factorizing it naively can be computationally expensive.

14.13 Global Word Vector Representations via Co-occurrence Statistics

Recall that our goal is to obtain a global vector representation for words, capturing semantic relationships beyond simple one-hot encodings. Instead of encoding words individually, we leverage *co-occurrence* statistics of word pairs within a corpus to build richer embeddings.

Setup: Consider two words w_i and w_j appearing in some context window within a text corpus. We are interested in modeling the *co-occurrence* of these words, possibly mediated by a third *context* word w_k. For example, in the phrase "big historic castle," the words "big" and "historic" are targets, and "castle" can be a context word connecting them.

Notation:

- Plain symbols w_i, w_j, w_k denote lexical items drawn from the vocabulary.
- Bold symbols denote vectors: $\mathbf{v}_i$ is the embedding of target word w_i and $\mathbf{u}_k$ the embedding of context word w_k.
- X_{ik} counts how often w_i and w_k co-occur within the chosen context window, and $X_i = \sum_k X_{ik}$ is the total number of context observations for w_i.

Goal: Define a function f that relates the co-occurrence statistics of the word pairs and context words to a scalar quantity representing their semantic associ-

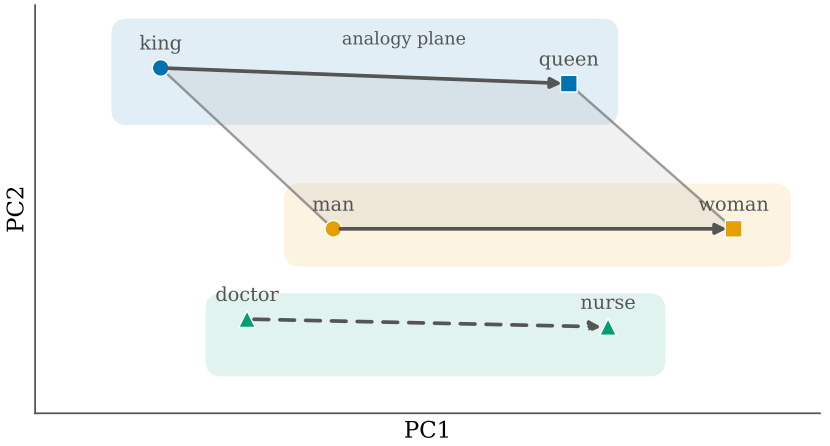

Figure 64: Schematic: Analogy geometry in embedding space. The classic offset "v(king) - v(man) + v(woman) approx v(queen)" forms a parallelogram; a similar gender direction also moves "doctor" toward "nurse." Visualizing these displacement vectors (solid vs. dashed) makes the shared relational direction explicit. Points are shown after a 2D PCA projection, so directions are approximate rather than exact.

ation.

Visualization. Projecting the learned vectors onto two principal components typically reveals well-separated semantic clusters. Figure 64 highlights how gendered titles, fruits, and locations occupy distinct regions, reinforcing that co-occurrence-driven training captures rich lexical structure.

14.13.1 Modeling Co-occurrence Probabilities

We start by considering the conditional probability of observing a context word w_k given a target word w_i:

$$P(k|i) = \frac{X_{ik}}{X_i}. \tag{14.13}$$

This probability captures how likely the context word w_k appears near the target word w_i.

Relating to word vectors: Suppose each word w_i is represented by a vector $\mathbf{v}_i \in \mathbb{R}^d$. We want to model the relationship between $\mathbf{v}_i$, $\mathbf{u}_k$, and the co-occurrence probability $P(k|i)$.

A natural assumption is that the co-occurrence probability can be modeled as an exponential function of the inner product of the corresponding word vectors:

$$P(k|i) \propto \exp\left(\mathbf{v}_i \, \mathbf{u}_k\right). \tag{14.14}$$

More explicitly, we can write a normalized model

$$P(k|i) \approx \frac{1}{Z_i} \exp\left(\mathbf{v}_i \, \mathbf{u}_k + b_i + b_k\right),$$

with partition function

$$Z_i = \sum_{k'} \exp\left(\mathbf{v}_i \, \mathbf{u}_{k'} + b_i + b_{k'}\right).$$

Taking logarithms on both sides and absorbing the (word-specific) normalizer into the biases gives the approximate relation

$$\log P(k|i) \approx \mathbf{v}_i \, \mathbf{u}_k + b_i + b_k, \tag{14.15}$$

where b_i and b_k are bias terms associated with words w_i and w_k, respectively. These biases account for the overall frequency or importance of each word while implicitly capturing the effect of Z_i.

Derivation: Starting from the co-occurrence counts,

$$\log X_{ik} - \log X_i = \log \frac{X_{ik}}{X_i} = \log P(k|i) \tag{14.16}$$

$$\approx \mathbf{v}_i \, \mathbf{u}_k + b_i + b_k. \tag{14.17}$$

This equation suggests that the log co-occurrence counts can be approximated by a bilinear form plus biases.

14.13.2 Optimization Objective

Given the corpus co-occurrence matrix $X = [X_{ik}]$, our goal is to find word vectors $\mathbf{v}_i$, $\mathbf{u}_k$ and biases b_i, b_k that minimize the reconstruction error:

$$J = \sum_{i,k} f(X_{ik}) \left(\mathbf{v}_i \, \mathbf{u}_k + b_i + b_k - \log X_{ik}\right)^2, \tag{14.18}$$

where f is a weighting function that controls the influence of each co-occurrence pair.

Why weighting? Many entries X_{ik} are zero or very small, which can cause numerical instability or dominate the objective. The function f is designed to:

- Downweight rare co-occurrences (small X_{ik}) to avoid overfitting noise.
- Possibly cap the influence of very frequent co-occurrences to prevent them from dominating.

A typical choice for f is:

$$f(x) = \begin{cases} \left(\frac{x}{x_{\max}}\right)^{\alpha} & \text{if } x < x_{\max}, \\ 1 & \text{otherwise,} \end{cases} \tag{14.19}$$

where $\alpha \in (0, 1)$ and $x_{\max}$ is a cutoff parameter.

14.13.3 Interpretation and Remarks

14.14 Finalizing the Word Embedding Derivations

In the previous sections, we explored the formulation of word embeddings through co-occurrence statistics and matrix factorization approaches. We now conclude the derivations and clarify the role of bias terms and optimization strategies.

Recall the key equation relating the word vectors $\mathbf{v}_i$ and context vectors $\mathbf{u}_k$ to the co-occurrence counts x_{ik}:

$$\mathbf{v}_i \, \mathbf{u}_k + b_i + b_k = \log x_{ik}, \tag{14.20}$$

where b_i and b_k are bias terms associated with the word and context, respectively.

Symmetry and Bias Terms Initially, two separate bias terms b_i and b_k were introduced to account for asymmetries in the data. However, it is often possible to simplify the model by combining or eliminating one of the biases without loss of generality. This is because the biases can absorb constant shifts in the embeddings, and the key information lies in the relative positions of the vectors. In practice we keep both biases so that very frequent terms (e.g., stop words)

can learn large offsets while rarer words keep their dot products numerically stable.

Hence, the equation can be rewritten as

$$\mathbf{v}_i \, \mathbf{u}_k = \log x_{ik} - b_i - b_k. \tag{14.21}$$

In practice, the biases b_i and b_k are learned jointly with the embeddings to best fit the observed co-occurrence statistics.

Objective Function and Optimization Let the target embeddings be rows $\mathbf{v}_i \in \mathbb{R}^{1\times d}$, the context embeddings be columns $\mathbf{u}_k \in \mathbb{R}^{d\times 1}$, and the biases scalars $b_i, b_k \in \mathbb{R}$. The scalar score $\mathbf{v}_i \, \mathbf{u}_k$ therefore measures the alignment between the target and context embeddings. The goal is to find $\{\mathbf{v}_i, \mathbf{u}_k, b_i, b_k\}$ that minimize the reconstruction error of the log co-occurrence matrix. Because raw counts span several orders of magnitude, the loss must behave like plain least squares for large x_{ik} yet dampen the influence of very small counts. Enforcing the limits $f(x) \to 0$ as $x \to 0$ and $f(x) \to 1$ for $x \geq x_{\max}$ yields the weighting scheme used by GloVe. The final weighted least-squares loss is

$$J = \sum_{i=1}^{V} \sum_{k=1}^{V} f(x_{ik}) \left(\mathbf{v}_i \, \mathbf{u}_k + b_i + b_k - \log x_{ik}\right)^2, \tag{14.22}$$

where $f(x)$ is the weighting function that downweights rare (or extremely common) co-occurrences to improve robustness. GloVe, for instance, uses the piecewise definition

$$f(x) = \begin{cases} \left(\frac{x}{x_{\max}}\right)^{\alpha} & \text{if } x < x_{\max}, \\ 1 & \text{otherwise,} \end{cases} \qquad 0 < \alpha \leq 1, \tag{14.23}$$

so that very small counts contribute little to the loss while still allowing moderately frequent pairs to influence the fit. In the original paper, practical defaults such as $\alpha \approx 0.75$ and $x_{\max} \approx 100$ were found to work well across a range of corpora (Pennington et al., 2014); keeping those guardrails explicit also explains why the same weighting recipe keeps reappearing in derived models.

Singular Value Decomposition (SVD) Connection One approach to solving this problem is to perform a low-rank approximation of the matrix $\log X$, where $X = [x_{ik}]$ is the co-occurrence matrix and the logarithm is applied elementwise (with small smoothing constants, e.g., $\epsilon = 10^{-8}$, added to avoid $\log 0$). The singular value decomposition (SVD) provides a principled method to find such a factorization:

$$\log X \approx U_r \Sigma_r V_r^\top, \tag{14.24}$$

where $U_r \in \mathbb{R}^{V \times r}$ and $V_r \in \mathbb{R}^{V \times r}$ contain the top-r singular vectors (for the desired embedding dimension $d = r$), and $\Sigma_r \in \mathbb{R}^{r \times r}$ is a diagonal matrix of the corresponding singular values. The truncation rank r, often between 100 and 300 in practice, acts exactly like the embedding dimensionality knob in neural models.

By setting

$$\mathbf{v}_i = (U_r)_i \Sigma_r^{1/2}, \quad \mathbf{u}_k = (V_r)_k \Sigma_r^{1/2},$$

we obtain embeddings that approximate the log co-occurrence matrix in a least-squares sense.

Interpretation and Limitations While SVD provides a closed-form solution, it does not explicitly model the bias terms b_i, b_k or the weighting function $f(x)$. Those additional degrees of freedom allow gradient-based methods such as GloVe to better match empirical co-occurrence ratios. Biases soak up unigram frequency effects while the weighting function prevents very noisy counts from dominating the fit.

14.15 Bias in Natural Language Processing

An important consideration in word embedding models is the presence of bias inherited from the training corpora. Since embeddings are learned from co-occurrence patterns in text, they reflect the statistical properties of the language data, including cultural and societal biases.

Sources of Bias - **Cultural Bias:** Text corpora often contain stereotypes or skewed representations of gender, ethnicity, and other social categories (e.g., news archives that associate "nurse" more frequently with women than men). - **Historical Bias:** Older texts may reflect outdated or prejudiced views. Dig-

itized literature from the 19th century, for instance, over-represents colonial perspectives. - **Language-Specific Bias:** Different languages and dialects encode different cultural norms and connotations, such as grammatical gender or honorifics that privilege particular groups.

Impact on Embeddings For example, the well-known analogy

$$\text{king} - \text{man} + \text{woman} \approx \text{queen}$$

illustrates that many embeddings support approximately linear semantic relationships. However, these same linear structures can also reveal problematic biases, such as associating certain professions or attributes disproportionately with one gender or group.

Debiasing Techniques Addressing bias in embeddings is an active area of research. Techniques include: - *Post-processing* embeddings to remove bias directions (e.g., Hard Debiasing by Bolukbasi et al. (2016)). - *Data augmentation* to balance training corpora or swap gendered terms. - *Regularization* during training to penalize biased associations or enforce equality constraints.

Cross-Lingual Challenges When extending embeddings to multiple languages, biases can manifest differently due to linguistic and cultural variations. For example, gender is grammatically encoded in Romance languages, so direct projection of English debiasing techniques may still leave gendered artifacts in Spanish or French embeddings. Careful consideration is required to ensure fairness and robustness across languages.

Practical bias checks

- **Dataset audit:** Inspect class balance, label sources, and sensitive attributes; check for under-represented groups and spurious correlations (e.g., profession ↔ gender cues).
- **Calibration and reliability:** Evaluate calibration (ECE, reliability diagrams) overall and for key subgroups; severely miscalibrated models magnify harm when used for decision support.
- **Disaggregated evaluation:** Report accuracy, ROC/PR, and calibration metrics by subgroup rather than only aggregate scores; look for systematic performance gaps.
- **Mitigation loop:** Combine data interventions (rebalancing, augmentation) with model-side debiasing and re-evaluation; treat mitigation as an iterative, experiment-driven process.

Author's note: embeddings mix geometry and bias

Embedding spaces faithfully capture geometry (analogies, clusters) precisely because they also capture the biases present in the data. Treat every downstream use as a combination of those two facets: audit the geometry you need, but also audit the offset directions you would rather suppress. Vector arithmetic makes biases quantifiable, so put that ability to work before shipping a model.

14.16 Responsible deployment checklist (appendix)

- **Purpose and consent.** Document the use-case, decision stakes, and where humans remain in the loop; distinguish exploratory prototypes from production decision aids.
- **Data lineage and licensing.** Track licenses for each corpus (newswire, Common Crawl, proprietary logs) and state whether downstream users may redistribute the embeddings or derived models.
- **Privacy and security.** Scan corpora for PII, redact when necessary, and restrict raw-data access. When embeddings leave the lab, accompany them with an acceptable-use policy and redaction guarantees.

- **Monitoring.** Deploy subgroup-aware metrics, calibration checks, and toxicity filters in production; log drifts and institute retraining/rollback thresholds.
- **Documentation.** Ship a short "model card" summarizing intended uses, failure modes, and evaluation data so downstream teams can reason about fit-for-purpose decisions.

14.17 Contextual embeddings and transformers

Static embeddings assign a single vector per word type, so polysemous words such as "bank" cannot adapt to their context. Transformer-based language models (e.g., BERT; Devlin et al., 2019) compute token representations conditioned on the entire sentence via multi-head self-attention, allowing each occurrence to carry a context-specific vector. The techniques developed in this chapter remain useful for lightweight models and as initialization, but modern NLP pipelines increasingly fine-tune contextual models to capture sentence-level nuance.

Wrap-up

In this chapter, we concluded the derivation of word embedding models based on co-occurrence statistics, emphasizing the role of bias terms and optimization strategies such as singular value decomposition. We highlighted the importance of understanding and mitigating bias in natural language processing, as embeddings inherently reflect the cultural and societal context of their training data. These considerations are crucial for developing fair and effective language models.

Key takeaways

- Word embeddings are dense vectors learned from co-occurrence statistics (local windows or global matrices).
- Analogies and clustering arise from linear geometry in the embedding space.
- Bias in corpora propagates to embeddings; debiasing and careful datasets are important.

Exercises and lab ideas

- Implement a minimal example from this chapter and visualize intermediate quantities (plots or diagnostics) to match the pseudocode.
- Stress-test a key hyperparameter or design choice discussed here and report the effect on validation performance or stability.
- Re-derive one core equation or update rule by hand and check it numerically against your implementation.

Where we head next. Chapter 15 steps away from neural embeddings to review the broader soft-computing toolkit (fuzzy logic, evolutionary ideas) that we previewed in the roadmap in Chapter 1; keep the fairness discussion in mind as we compare alternative paradigms for reasoning under uncertainty.

References. Full citations for works mentioned in this chapter appear in the book-wide bibliography.

Part IV: Soft computing and fuzzy reasoning

15 Introduction to Soft Computing

Learning Outcomes

- Articulate why soft computing (fuzzy logic, evolutionary search, neural hybrids) complements the statistical strand from earlier chapters.
- Define fuzzy sets, linguistic variables, and rule bases at a conceptual level before Chapters 16 to 18 formalize them.
- Track a running thermostat/autofocus example that grounds the design choices introduced throughout the fuzzy trilogy.

Running example: fuzzy thermostat

We revisit a smart thermostat that regulates a room using two linguistic inputs (temperature error and rate of change) and one output (heater power). This compact scenario lets us instantiate membership functions (Chapter 16), transform them between universes (Chapter 17), and assemble complete inference systems (Chapter 18) without inventing new notation each time.

Building on Chapter 14, which concluded the neural (biologically inspired) strand with modern language models and their practical pitfalls, we now pivot to the behavioral and evolutionary strand previewed in the roadmap in Chapter 1: soft computing. The roadmap figure (Figure 1) marks this transition explicitly.

Design motif

When precision is costly or ill-defined, make the vagueness explicit and keep the system auditable: represent linguistic concepts with membership functions, reason with operator choices you can explain, and tune those choices with optimization rather than brittle rules.

15.1 Hard Computing: The Classical Paradigm

Hard computing refers to the classical approach to computation where the goal is to produce precise, unambiguous, and mathematically exact outputs. This paradigm assumes that the relationships between inputs and outputs can be modeled accurately using well-defined mathematical equations. For example, Einstein's mass-energy equivalence formula,

$$E = mc^2, \tag{15.1}$$

is a precise, unambiguous, and exact mathematical expression.

In hard computing, the process typically involves:

- Precise inputs,
- Deterministic models,
- Exact outputs.

However, this approach is often inadequate for many real-world problems because:

1. The real world is pervasively *imprecise* and *uncertain*.
2. Achieving precision and certainty is often *costly* and *difficult*.

These limitations motivate the need for alternative computational frameworks that can tolerate and exploit imprecision and uncertainty.

15.2 Soft Computing: Motivation and Definition

Soft computing, introduced by Zadeh (1994, 1997) after his 1965 fuzzy sets paper, is a computational paradigm designed to handle problems where precision and certainty are either impossible or prohibitively expensive to obtain. Unlike hard computing, soft computing tolerates *imprecision*, *uncertainty*, and *approximate reasoning* to achieve solutions that are:

- **Tractable:** Computationally feasible to obtain,
- **Robust:** Insensitive to noise and variations,
- **Low-cost:** Economical in terms of computational resources.

Formally, soft computing is not a single homogeneous methodology but rather a *partnership of distinct methods* that conform to these guiding principles.

In Zadeh's broad usage, the principal constituents include:

- **Fuzzy Logic:** Handling imprecision and approximate reasoning,
- **Neurocomputing (and neuro-fuzzy hybrids):** Learning from data through neural networks, sometimes combined with fuzzy rule bases (e.g., ANFIS),
- **Genetic Algorithms:** Evolutionary optimization inspired by natural selection.

These components often overlap and complement each other in practical applications. In this book, probabilistic modeling is treated in the statistical strand (Chapters 3 to 4); the soft-computing block focuses on fuzzy systems and evolutionary search, with neuro-fuzzy hybrids serving as the bridge back to the neural chapters.

15.3 Why Soft Computing?

The key insight behind soft computing is to exploit the *tolerance for imprecision and uncertainty* inherent in many real-world problems. Consider the example of handwritten digit recognition using a convolutional neural network (CNN):

- The input is a handwritten digit, say the digit "4".
- The network extracts features and produces a probability distribution over possible digits.
- The output might be:

$$P(\text{digit} = 4) = 0.60, \quad P(\text{digit} = 7) = 0.20, \quad P(\text{digit} = 1) = 0.20.$$

This output is *not precise* in the classical sense; it expresses uncertainty and partial belief. The system tolerates this imprecision and still makes a decision based on the highest probability, demonstrating robustness and flexibility.

15.4 Relationship Between Hard and Soft Computing

We can conceptualize the landscape of computing as follows:

- **Hard Computing:** Precise, deterministic, mathematically exact.

- **Soft Computing:** Approximate, tolerant of imprecision and uncertainty, heuristic.

There is some overlap, especially in optimization problems, which can be approached via either paradigm depending on the context and requirements.

15.5 Overview of Soft Computing Constituents

Fuzzy Logic: Deals with *fuzziness* or vagueness, allowing partial membership in sets and approximate reasoning. It is particularly useful when information is incomplete or linguistic in nature.

Neurocomputing: Encompasses various neural network architectures (multilayer perceptrons, convolutional networks, recurrent models, Hopfield networks, and Radial Basis Function (RBF) networks) as well as neuromorphic hardware that learn from data and approximate complex nonlinear mappings.

Probabilistic Reasoning: Manages uncertainty using probability theory, belief networks, and Bayesian inference. It assumes known or estimable probability distributions.

Genetic Algorithms: Inspired by biological evolution, these algorithms perform heuristic search and optimization by mimicking natural selection and genetic variation.

15.6 Distinguishing Imprecision, Uncertainty, and Fuzziness

It is important to clarify the subtle differences among these concepts:

- **Uncertainty** refers to situations where the outcome is unknown but can be described probabilistically. For example, a classifier might assign a 60% probability to a particular class.
- **Imprecision** refers to limited resolution or vagueness in the available descriptions or measurements. Saying that the outside temperature is "warm" rather than specifying 24.5°C is imprecise because we are unsure about the precise boundary that should separate "warm" from "hot."
- **Fuzziness** captures graded membership in a linguistic category; for instance, the extent to which a day is "warm." Membership values range

continuously between 0 and 1 instead of forcing a binary decision.

In short, imprecision concerns our knowledge about a precise boundary, whereas fuzziness is a property of the concept itself: even with perfect measurements, "warm" transitions smoothly into "hot." For example, reading 24.5°C from a thermometer with ±1°C resolution is an *imprecise* observation, whereas deciding whether 24.5°C should be labelled "warm" or "hot" is a *fuzzy* membership question that remains even if the thermometer were infinitely precise.

Imprecision vs. Fuzziness

Imprecision concerns uncertainty about the exact value or boundary (e.g., measurement error or coarse resolution). **Fuzziness** concerns graded membership in a concept (e.g., the degree to which a day is "warm") even when measurements are exact. Probability quantifies uncertainty about events; fuzziness quantifies degree of truth of linguistic predicates.

15.7 Soft Computing: Motivation and Overview

Soft computing is not a monolithic framework but rather a coalition of distinct methods unified by a common goal: to exploit tolerance for imprecision, uncertainty, and partial truth to achieve tractability, robustness, and low solution cost. Unlike traditional hard computing, which demands exact inputs and produces precise outputs, soft computing embraces the inherent vagueness of many real-world problems, particularly those involving human reasoning and perception. The constituents mirror the probabilistic and connectionist tools from Chapters 3 to 14 but favour interpretability and rule-based reasoning:

- **Fuzzy Logic:** Captures human knowledge and reasoning expressed in linguistic terms, allowing approximate reasoning with imprecise concepts.
- **Neurocomputing (Neural Networks):** Learning from data and pattern recognition; hybrids such as ANFIS (Jang, 1993) blend fuzzy rules with trainable neural layers.
- **Probabilistic modeling (already covered):** Bayesian/MAP views from Chapters 3 to 4 remain complementary to fuzzy possibility views (Dubois and Prade, 1988), but the emphasis in this block is on rule-based reasoning rather than probabilistic inference.

Table 6: Schematic: Fuzzy vs. probabilistic reasoning at a glance.

	Fuzzy logic	**Probabilistic logic**
Semantics	Degree of membership (vagueness); e.g., "temperature is high to degree 0.7."	Degree of belief/uncertainty—probability that an event occurs.
Operators	t-norms / s-norms (min, product, max) model AND/OR; implication via fuzzy rules.	Sum / product rules, Bayes' theorem govern AND/OR/conditionals.
Outputs	Fuzzy sets defuzzified to crisp actions (e.g., heater power).	Numeric probabilities used for expectation, decision thresholds.
Typical use	Rule bases, approximate control, linguistic policies.	Stochastic modeling, hypothesis testing, Bayesian inference.

Table 7: Schematic: Boolean operators vs. fuzzy operators at a glance.

	Boolean logic	**Fuzzy logic**
AND	$\min(a, b)$	t-norm (e.g., min, product)
OR	$\max(a, b)$	s-norm (e.g., max, prob. sum)
NOT	$1 - a$	complement $1 - a$

- **Genetic/Evolutionary Computation:** Chapter 19 shows how evolutionary search tunes rule bases and membership parameters (Herrera and Lozano, 2008; Ishibuchi and Nakashima, 2007).

15.8 Fuzzy Logic: Capturing Human Knowledge Linguistically

One of the most compelling aspects of fuzzy logic is its ability to represent human knowledge and experience in a linguistic form that machines can process. Consider the everyday reasoning:

If you wake up late and the traffic is congested, then you will be late.

This statement involves vague concepts such as "late," "congested," and "will be late," which are not crisply defined but are intuitively understood by

humans. Fuzzy logic allows us to formalize such rules without requiring precise probabilistic models or extensive training data.

Fuzzy Rules and Approximate Reasoning A fuzzy rule typically has the form:

$$\text{IF } A \text{ AND } B \text{ THEN } C, \tag{15.2}$$

where A, B, and C are fuzzy propositions characterized by membership functions rather than crisp sets.

For example:

- A: "Wake up late" could be represented by a membership function $\mu_{\text{late}}(t)$ over the waking time t.
- B: "Traffic is congested" could be represented by a membership function $\mu_{\text{congested}}(x)$ over traffic density x.
- C: "You will be late" is the fuzzy output.

Each membership function maps from the relevant universe of discourse to $[0, 1]$, i.e., $\mu_{\text{late}} : \mathbb{R} \to [0, 1]$, so that linguistic labels become numeric degrees of support. The fuzzy inference system combines these membership values using *t-norm* operators (e.g., min, product) to model logical conjunction and *s-norms* (e.g., max) to model disjunction, thereby inferring the degree to which the conclusion C holds. In practical systems the resulting fuzzy set is often *defuzzified* (e.g., via centroid or maximum-membership methods) to obtain a single crisp recommendation.

Advantages over Traditional Systems Traditional rule-based systems or statistical models require precise numerical inputs or probability distributions. In contrast, fuzzy logic:

- Does not require exact numerical data or probability distributions.
- Allows direct encoding of expert knowledge in natural language.
- Handles imprecision and vagueness inherent in human concepts.
- Provides interpretable models that align with human reasoning.

15.9 Comparison with Other Soft Computing Paradigms

Neural Networks Neural networks model complex nonlinear relationships by learning from data. They transform input features $\mathbf{x} \in \mathbb{R}^n$ into new feature spaces through weighted sums and nonlinear activations:

$$\mathbf{h} = \sigma(\mathbf{W}^\top \mathbf{x} + \mathbf{b}), \tag{15.3}$$

where $\mathbf{W} \in \mathbb{R}^{n \times m}$ maps the n-dimensional input into an m-dimensional hidden space, $\mathbf{b} \in \mathbb{R}^m$ is the bias vector, and $\sigma(\cdot)$ is a nonlinear activation function applied elementwise.

Unlike fuzzy logic, neural networks require training on large datasets and do not inherently provide interpretable linguistic rules; there is, however, an active line of research on *rule extraction* and network distillation aimed at recovering approximate linguistic descriptions from trained models.

Genetic Algorithms Genetic algorithms simulate evolutionary processes to optimize solutions by iteratively selecting, recombining, and mutating candidate solutions. They are useful for derivative-free optimization and problems with complex search spaces.

Probabilistic Reasoning Probabilistic methods model uncertainty explicitly using probability distributions and Bayesian inference. They require knowledge or estimation of underlying distributions, which may be difficult in many practical scenarios, but approximate inference schemes (e.g., Monte Carlo sampling, variational methods) can mitigate this requirement when exact distributions are unavailable.

15.10 Zadeh's Insight and the Birth of Fuzzy Logic

Lotfi Zadeh, in the late 1960s, observed that classical statistics and probability theory demand precise knowledge of distributions and exact calculations, which is often unrealistic for human decision-making. Humans rely on approximate, linguistic knowledge rather than exact numerical data.

Zadeh's key insight was to develop a mathematical framework that could:

- Represent imprecise concepts using fuzzy sets.

- Allow approximate reasoning with these fuzzy sets.
- Enable machines to operate based on human-like linguistic rules.

This approach revolutionized how we model uncertainty and reasoning in artificial intelligence and control systems.

15.11 Challenges in Fuzzy Logic Systems

Despite its advantages, fuzzy logic faces several challenges:

- **Lack of a systematic methodology:** Initially, there was no formal mechanism to construct fuzzy inference systems from human knowledge.
- **Handling imprecision in linguistic terms:** Choosing membership functions and linguistic labels still relies on expert elicitation or data-driven tuning; poor choices can degrade system performance.

15.12 Mathematical Languages as Foundations for Fuzzy Logic

Recall that the motivation behind fuzzy logic was to develop a mathematical and linguistic framework capable of handling imprecision and uncertainty in a principled way. To achieve this, Lotfi Zadeh drew inspiration from several well-established mathematical languages, each with its own syntax, semantics, and rules of inference. Understanding these languages helps us appreciate how fuzzy logic extends and generalizes classical logic to accommodate vagueness.

15.12.1 Relational Algebra

Relational algebra is a formal language used primarily in database theory to manipulate sets and relations. It provides operators such as union ($\cup$), intersection ($\cap$), and set difference ($\setminus$) that operate on sets:

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\}, \tag{15.4}$$

$$A \cap B = \{x \mid x \in A \text{ and } x \in B\}. \tag{15.5}$$

The third canonical operator is the set difference

$$A \setminus B = \{x \mid x \in A \text{ and } x \notin B\},$$

which removes from A any elements that also belong to B. For instance, if A is the set of all graduate students and B the set of teaching assistants, then $A \setminus B$ contains graduate students who are not currently TAs.

These operators have well-defined meanings and predictable outputs, making relational algebra a precise language for reasoning about collections of elements. The vocabulary is limited but sufficient for set-theoretic operations.

15.12.2 Boolean Algebra

Boolean algebra is the algebraic structure underlying classical logic and digital circuits. It operates on binary variables taking values in $\{0, 1\}$, with logical operators such as `AND` ($\wedge$), `OR` ($\vee$), and `XOR` ($\oplus$):

$$A \vee B = 1 \quad \text{if } A = 1 \text{ or } B = 1, \tag{15.6}$$

$$A \wedge B = 1 \quad \text{if } A = 1 \text{ and } B = 1, \tag{15.7}$$

$$A \oplus B = 1 \quad \text{if } A \neq B. \tag{15.8}$$

Conversely, $A \vee B = 0$ only when both inputs are 0, and $A \wedge B = 0$ unless both inputs equal 1; the XOR operator returns 0 exactly when both operands share the same truth value.

Boolean algebra provides a crisp, binary framework where propositions are either true or false, with no intermediate values. This crispness is a limitation when modeling real-world phenomena involving gradations of truth.

15.12.3 Predicate Algebra

Predicate algebra extends Boolean algebra by incorporating quantifiers and variables, allowing statements about properties of elements in a domain. For example, a predicate statement might be:

$$\forall x \in \mathbb{R}, \quad x^2 \geq 0,$$

which reads: "For all real numbers x, x^2 is greater than or equal to zero." This language combines logical connectives with quantifiers such as $\forall$ (for all) and $\exists$ (there exists), enabling more expressive statements about sets and relations.

An example involving two domains could be:

$$\forall x \in \text{Rabbits}, \quad \forall y \in \text{Tortoises}, \quad \text{Faster}(x, y),$$

meaning "For any rabbit x and any tortoise y, x is faster than y."

Predicate algebra thus provides a linguistic and symbolic framework to express complex relationships and properties.

15.12.4 Propositional Calculus

Propositional calculus (or propositional logic) deals with propositions and their logical connectives. It focuses on the relationships between propositions without internal structure. The basic form involves premises and conclusions, such as:

$$P \implies Q, \quad P \quad \Rightarrow \quad Q, \tag{15.9}$$

where P and Q are propositions, and $\implies$ denotes implication.

Modus Ponens One fundamental rule of inference in propositional calculus is *modus ponens*:

If $P \implies Q$ and P is true, then Q must be true.

Symbolically,

$$P \implies Q, \quad P \quad \vdash \quad Q. \tag{15.10}$$

This rule affirms the consequent by affirming the antecedent.

Modus Tollens Another inference rule is *modus tollens*:

If $P \implies Q$ and Q is false, then P must be false.

Symbolically,

$$P \implies Q, \quad \neg Q \quad \vdash \quad \neg P. \tag{15.11}$$

This rule denies the antecedent by denying the consequent. However, as noted, this inference can sometimes be risky or invalid in practical scenarios due to exceptions or additional factors.

Hypothetical Syllogism A further inference pattern is the *hypothetical syllogism*:

If $P \implies Q$ and $Q \implies R$, then $P \implies R$.

Symbolically,

$$P \implies Q, \quad Q \implies R \quad \vdash \quad P \implies R. \tag{15.12}$$

This transitive property of implication allows chaining of logical statements.

15.13 Fuzzy Logic as a New Mathematical Language

Zadeh's insight was to synthesize these classical mathematical languages into a new framework that could handle degrees of truth rather than binary true/false values. Fuzzy logic generalizes Boolean algebra by allowing truth values to range continuously over the interval $[0, 1]$, representing partial truth

15.14 Fuzzy Logic: Motivation and Intuition

Recall that classical (crisp) logic deals with binary truth values: a proposition is either true (1) or false (0). For example, the question "Was the exam easy?" can be answered crisply as "Yes" or "No." However, many real-world situations are not so black-and-white. Often, we want to express uncertainty or partial truth, such as "The exam was somewhat easy," or "The exam was easy to a certain degree."

Fuzzy truth values allow us to express such intermediate degrees of truth. Instead of restricting truth values to $\{0, 1\}$, fuzzy logic permits any value in the continuous interval $[0, 1]$. For instance, if the exam was moderately easy, we might assign a truth value of 0.6 or 0.7, indicating partial truth.

This flexibility captures the inherent vagueness in many human concepts and perceptions. For example, when asked "Did you enjoy your lunch?" one might respond "sort of," reflecting a fuzzy assessment rather than a crisp yes/no.

Why fuzzy logic?

- **Tolerance for imprecision:** Observations and measurements are often noisy or uncertain.
- **Expressiveness:** Allows linguistic hedging such as "somewhat," "maybe," or "approximately."
- **Robustness:** Systems can handle ambiguous or incomplete information gracefully.

15.15 From Crisp Sets to Fuzzy Sets

Crisp sets are classical sets where an element either belongs or does not belong to the set. Formally, for a universe X, a crisp set $A \subseteq X$ is characterized by its *characteristic function*:

$$\chi_A(x) = \begin{cases} 1 & \text{if } x \in A, \\ 0 & \text{if } x \notin A. \end{cases}$$

Example: Consider two classes:

$$\text{Class 1} = \{\text{Li}, \text{Rajnish}\}, \quad \text{Class 2} = \{\text{Hamid}, \text{John}, \text{Julia}, \text{Yet}\}.$$

These are crisp sets since no student belongs to both classes simultaneously.

Fuzzy sets generalize this notion by allowing partial membership. A fuzzy set $\tilde{A}$ on X is characterized by a *membership function*:

$$\mu_{\tilde{A}} : X \to [0, 1],$$

where $\mu_{\tilde{A}}(x)$ quantifies the degree to which x belongs to $\tilde{A}$.

Example: Sizes as fuzzy sets

Consider the linguistic labels `Small`, `Medium`, and `Large` for weights (in kilograms). A crisp partition such as $[0, 10], [11, 20], [21, 30]$ is disjoint; fuzzy sets allow these labels to overlap smoothly so a weight can belong to both `Medium` and `Large` to different degrees. See Chapter 16 (especially Section 16.7) for the explicit membership formulas and plots; here keep the intuition that fuzzy labels overlap and map a universe of discourse into $[0, 1]$.

Thermostat at a glance. Throughout Chapters 16 to 18 we reuse a fuzzy thermostat: inputs are temperature error and rate (linguistic labels such as `Cold`, `Warm`, `Hot`; `Cooling`, `Stable`, `Heating`); rules map these to heater power; defuzzification turns the fuzzy action into a crisp control signal. Keep this loop in mind as membership functions and operators are introduced.

Lab prep: fuzzy thermostat starter

- Install `scikit-fuzzy` and `matplotlib`.
- Define triangular membership functions for `Cold`/`Warm`/`Hot` over a temperature universe; plot the overlap.
- Write one rule: IF error is `Cold` AND rate is `Heating` THEN power is `Low`; preview centroid defuzzification.

Exercises (Chapter 15)

- Classify three scenarios as imprecision vs. uncertainty vs. fuzziness; justify each.
- Write two fuzzy thermostat rules and reason qualitatively about the output for a borderline input.
- Compare min vs. product t-norm on the same antecedent degrees (e.g., 0.4 and 0.7); explain the impact.
- Sketch (or code) a triangular membership and a simple IF–THEN rule; describe how defuzzification would proceed.
- Identify where probability (Chapter 3) and fuzzy possibility (this chapter) would lead to different interpretations.

Forward pointer. Chapter 16 builds the membership functions and universes for the thermostat inputs/outputs; Chapters 17 to 18 assemble full inference and defuzzification, and Chapter 19 shows how evolutionary search can tune rule bases and memberships.

15.16 Wrapping Up Fuzzy Sets and Fuzzy Logic

In this final part of the chapter, we conclude our introduction to fuzzy sets and fuzzy logic by summarizing key concepts and clarifying the open points from the

previous discussion.

Fuzzy Sets Recap Recall that a *fuzzy set* A defined on a universe of discourse X is characterized by a *membership function*

$$\mu_A : X \to [0, 1],$$

which assigns to each element $x \in X$ a degree of membership $\mu_A(x)$ indicating the extent to which x belongs to the set A. Unlike classical (crisp) sets where membership is binary (0 or 1), fuzzy sets allow partial membership, capturing the inherent vagueness of many real-world concepts.

Universe of Discourse The *universe of discourse* X is the domain over which fuzzy sets are defined. For example, if X represents the set of all students, fuzzy subsets could be "tall students," "medium height students," and "short students," each with overlapping membership functions reflecting the subjective nature of these categories.

Fuzziness and Degrees of Truth Fuzzy logic extends classical Boolean logic by allowing truth values to range continuously between 0 and 1. This enables reasoning with imprecise or approximate information, such as the statement "the water is warm," which is neither absolutely true nor false but has a degree of truthfulness.

Example: Height Classification Consider the linguistic variables "short," "medium," and "tall." In classical logic, a person is either short or not, tall or not, with crisp boundaries. In fuzzy logic, these categories overlap, and a person's height can partially belong to multiple categories simultaneously. This reflects human intuition and natural language better than crisp sets.

Fuzzy Actions and Control In intelligent control systems, such as automotive braking, fuzzy logic allows the control actions to be fuzzy themselves. Instead of a binary decision to "hit the brakes" or "not hit the brakes," the system can decide to apply the brakes "somewhat," "moderately," or "strongly,"

based on fuzzy inputs like distance and speed. This leads to smoother, more adaptive control.

Next Steps: Membership Functions and Fuzzy Inference Systems Chapters 16 to 18 pick up the thermostat running example and formalize each stage: Chapter 16 constructs the membership functions for error/rate labels, Chapter 17 shows how relations and projections move information between universes, and Chapter 18 assembles the full Mamdani/Sugeno inference pipeline. Keep this soft-computing overview handy as a conceptual map while those chapters work through the algebra.

Key takeaways

- Soft computing embraces imprecision via fuzzy logic, evolutionary search, and neural networks.
- Fuzzy operators (t-norms, implications) enable approximate reasoning under uncertainty.
- Choosing operators and membership functions matches problem semantics to inference behavior.

Exercises and lab ideas

- Implement a minimal example from this chapter and visualize intermediate quantities (plots or diagnostics) to match the pseudocode.
- Stress-test a key hyperparameter or design choice discussed here and report the effect on validation performance or stability.
- Re-derive one core equation or update rule by hand and check it numerically against your implementation.

Where we head next. Chapters 16 to 18 develop the thermostat running example end-to-end: Chapter 16 builds membership functions, Chapter 17 moves information between universes via relations, and Chapter 18 assembles full inference and defuzzification.

References. Full citations for works mentioned in this chapter appear in the book-wide bibliography.

16 Fuzzy Sets and Membership Functions: Foundations and Representations

Building on Chapter 15, we now formalize the foundations of fuzzy logic: universes of discourse, fuzzy sets, membership functions, and the basic operators used throughout the fuzzy trilogy. These tools define the linguistic labels for the running thermostat example and prepare the ground for fuzzy relations and full inference systems in Chapters 17 to 18. The roadmap in Figure 1 situates this as the start of the fuzzy trilogy.

Design motif

Treat vagueness as a first-class modeling choice: write the membership functions down, pick operators explicitly, and then audit how those choices shape the behavior of a rule base.

Learning Outcomes

After this chapter, you should be able to:

- Distinguish imprecision (uncertain value/boundary) from fuzziness (graded membership).
- Define and interpret membership functions in discrete and continuous domains.
- Apply fuzzy set operations and De Morgan's laws using max/min forms.
- Execute an end-to-end Mamdani inference and compute the centroid defuzzification.

Running example checkpoint

For the thermostat scenario introduced in Chapter 15, the universe of discourse for the inputs is $[-5, 5]$°C temperature error and $[-2, 2]$°C/min rate-of-change. As you study triangular, trapezoidal, and Gaussian membership functions, imagine parameterizing the linguistic labels *Cold*, *Comfy*, and *Hot* for these inputs. We reuse those shapes when composing rules in Chapters 17 to 18.

Operator defaults used in Chapters 16 to 18

Unless stated otherwise, the fuzzy trilogy uses the *standard* operators:

- Complement (negation): $C(\mu) = 1 - \mu$ (so $C(C(\mu)) = \mu$).
- Intersection and union: $T_{\min}(a, b) = \min(a, b)$, $S_{\max}(a, b) = \max(a, b)$.
- De Morgan's laws are interpreted with this standard complement.

Alternative t-/s-norms or complements are called out explicitly when they appear.

16.1 Recap: Fuzzy Sets and the Universe of Discourse

Recall that a *fuzzy set* A in a universe of discourse X is characterized by a *membership function* $\mu_A : X \to [0, 1]$. This membership function assigns to each element $x \in X$ a degree of membership $\mu_A(x)$, which quantifies the extent to which x belongs to the fuzzy set A.

- If $\mu_A(x) = 1$, then x fully belongs to A.
- If $\mu_A(x) = 0$, then x does not belong to A at all.
- If $0 < \mu_A(x) < 1$, then x partially belongs to A to the degree $\mu_A(x)$.

This contrasts with classical (crisp) sets, where membership is binary (either 0 or 1).

16.2 Membership Functions: Definition and Interpretation

A *membership function* $\mu_A(x)$ maps each element x in the universe X to a membership grade in the interval $[0, 1]$. The shape and parameters of μ_A encode the fuzziness or uncertainty associated with the concept represented by A.

Example: Consider the fuzzy set *Slow Speed* defined over the universe of speeds $X \subseteq \mathbb{R}$. The membership function $\mu_{\text{Slow}}(x)$ might assign high membership values to speeds near 20 km/h and gradually decrease as speed increases, reflecting the gradual transition from "slow" to "not slow."

Mathematical Representation: For each $x \in X$,

$$\mu_A(x) \in [0, 1]. \tag{16.1}$$

The fuzzy set A can be represented as the collection of ordered pairs:

$$A = \{(x, \mu_A(x)) \mid x \in X\}. \tag{16.2}$$

16.3 Discrete vs. Continuous Universes of Discourse

The universe X can be either discrete or continuous, which affects how fuzzy sets and membership functions are represented.

16.3.1 Discrete Universe

When $X = \{x_1, x_2, \ldots, x_n\}$ is finite or countable, the fuzzy set A is represented as a finite collection of ordered pairs:

$$A = \{(x_1, \mu_A(x_1)), (x_2, \mu_A(x_2)), \ldots, (x_n, \mu_A(x_n))\}. \tag{16.3}$$

Typically, membership values equal to zero are omitted for brevity, since they indicate no membership.

Example: Suppose $X = \{1, 2, 3, 4, 5\}$ and the membership function values are:

$$\mu_A(1) = 0, \quad \mu_A(2) = 0.1, \quad \mu_A(3) = 0.3, \quad \mu_A(4) = 0.7, \quad \mu_A(5) = 0.$$

Then,

$$A = \{(2, 0.1), (3, 0.3), (4, 0.7)\}.$$

16.3.2 Continuous Universe

When $X \subseteq \mathbb{R}$ is continuous (e.g., an interval), the fuzzy set A is described by a membership function $\mu_A(x)$ defined for all $x \in X$. The representation is

functional rather than enumerative:

$$A = \int_{x \in X} \mu_A(x)/x, \tag{16.4}$$

where the notation $\int \mu_A(x)/x$ denotes the continuous collection of pairs $(x, \mu_A(x))$.

Interpretation: The integral sign here is symbolic, indicating a continuous aggregation over X, not a numerical integral in the calculus sense.

Example: Consider a triangular membership function centered at c with base width w:

$$\mu_A(x) = \max\left(0, 1 - \frac{|x - c|}{w}\right). \tag{16.5}$$

This function assigns membership 1 at $x = c$, decreasing linearly to zero at $x = c \pm w$.

16.4 Crisp Sets versus Fuzzy Sets

Crisp (classical) sets assign membership values in the binary set $\{0, 1\}$, so each element either belongs to the set or it does not. In contrast, fuzzy sets allow intermediate membership values, enabling gradual transitions between full inclusion and full exclusion. Understanding this contrast highlights why membership functions are central to fuzzy logic.

Imprecision vs. Fuzziness (recap)

As discussed in Section 15.6, **imprecision** concerns uncertainty about the exact value or boundary (e.g., measurement noise or coarse resolution), whereas **fuzziness** concerns graded membership in a concept (e.g., the degree to which a speed is "slow") even when measurements are exact. Probability models uncertainty about events; fuzzy logic models degrees of truth of linguistic predicates.

16.5 Membership Functions in Fuzzy Sets

Recall that a fuzzy set A on a universe X is characterized by a membership function $\mu_A : X \to [0, 1]$ which assigns to each element $x \in X$ a degree of

membership $\mu_A(x)$ indicating the extent to which x belongs to A.

Triangular Membership Function One of the simplest and most intuitive membership functions is the *triangular* membership function. It is defined by three parameters $a < b < c$ and given by

$$\mu_A(x) = \begin{cases} 0, & x \leq a \\ \frac{x-a}{b-a}, & a < x \leq b \\ \frac{c-x}{c-b}, & b < x < c \\ 0, & x \geq c \end{cases} \tag{16.6}$$

This function attains its maximum value 1 at $x = b$, representing the point of highest confidence that x belongs to the fuzzy set A. The membership decreases linearly on either side of b, reaching zero at a and c. This shape expresses a strong belief in membership near b and uncertainty elsewhere.

Trapezoidal Membership Function The trapezoidal membership function generalizes the triangular shape by allowing a flat top, representing a range of values with full membership. It is defined by four parameters $a < b \leq c < d$:

$$\mu_A(x) = \begin{cases} 0, & x \leq a \\ \frac{x-a}{b-a}, & a < x \leq b \\ 1, & b < x \leq c \\ \frac{d-x}{d-c}, & c < x < d \\ 0, & x \geq d \end{cases} \tag{16.7}$$

This function models situations where there is full confidence that all values between b and c belong to the fuzzy set, with gradual transitions on the edges.

Gaussian Membership Function The Gaussian membership function is widely used due to its smoothness and differentiability, which are advantageous in optimization and learning algorithms. It is defined by parameters c (center) and $\sigma > 0$ (width):

$$\mu_A(x) = \exp\left(-\frac{(x-c)^2}{2\sigma^2}\right). \tag{16.8}$$

This bell-shaped curve smoothly assigns membership values, with the highest membership at $x = c$ and decreasing membership as x moves away from c. The parameter σ controls the spread or fuzziness of the set.

Generalized Bell Membership Function Another flexible membership function is the generalized bell function, defined by parameters a, b, c:

$$\mu_A(x) = \frac{1}{1 + \left|\frac{x-c}{a}\right|^{2b}}. \tag{16.9}$$

This function allows control over the width and slope of the membership curve, interpolating between shapes similar to triangular and Gaussian functions.

16.6 Comparison of Membership Functions

- **Triangular and Trapezoidal:** These are piecewise linear, computationally inexpensive, and easy to interpret. However, they are not differentiable at the vertices, which can be a limitation in gradient-based learning.
- **Gaussian and Bell:** These are smooth and differentiable, making them suitable for optimization and adaptive systems. They provide more modeling flexibility but are computationally more expensive.

Example: Grading System as Fuzzy Sets Consider a typical university grading scale as an example of fuzzy sets. Traditional crisp sets assign grades as follows:

$$\text{F} : [0, 59], \quad \text{D} : [60, 69], \quad \text{C} : [70, 79], \quad \text{B} : [80, 89], \quad \text{A} : [90, 100].$$

In a crisp set, membership is binary: a score of 75 is fully in C and not in B.

However, students and instructors may perceive these boundaries differently. For example, some may consider 75 to be a borderline B, or 68 to be a borderline C. This uncertainty can be modeled by fuzzy sets with overlapping membership functions.

For instance, the membership function for grade C could be trapezoidal:

$$\mu_C(x) = \begin{cases} 0, & x \le 65, \\ \dfrac{x-65}{5}, & 65 < x \le 70, \\ 1, & 70 < x \le 75, \\ \dfrac{80-x}{5}, & 75 < x \le 80, \\ 0, & x > 80. \end{cases}$$

Similarly, the membership for grade B could be written as

$$\mu_B(x) = \begin{cases} 0, & x \le 75, \\ \dfrac{x-75}{5}, & 75 < x \le 80, \\ 1, & 80 < x \le 85, \\ \dfrac{90-x}{5}, & 85 < x \le 90, \\ 0, & x > 90, \end{cases}$$

with analogous expr apping trape-
zoids capture the ir multaneously
belong to both C a e 65.

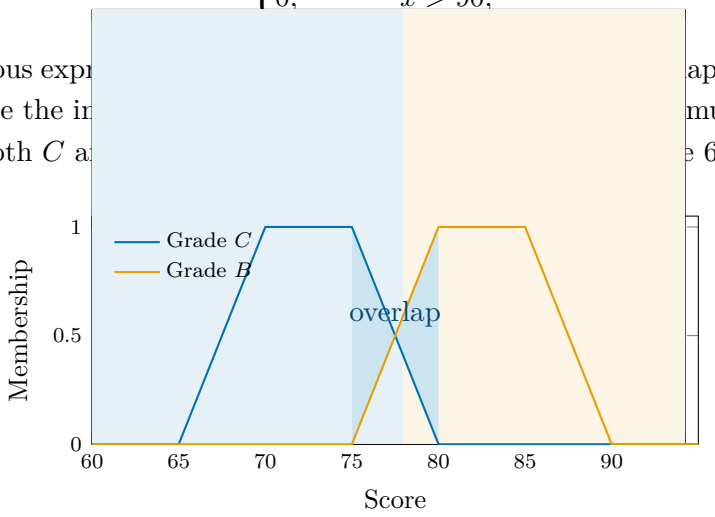

Figure 65: Schematic: Trapezoidal membership functions for grades C and B with the overlapping region shaded. Scores near 78–82 partially satisfy both grade definitions.

16.7 Example: Overlapping weight labels

Fuzzy labels often overlap so that borderline values belong to multiple sets. For weights measured in kilograms, one crisp partition is $[0, 10], [11, 20], [21, 30]$ for `Small`, `Medium`, `Large`. A fuzzy partition smooths these transitions:

$$\mu_{\text{Small}}(x) = \begin{cases} 1, & x \le 10, \\ 1 - \dfrac{x - 10}{5}, & 10 < x < 15, \\ 0, & x \ge 15, \end{cases}$$

$$\mu_{\text{Medium}}(x) = \begin{cases} 0, & x \le 10, \\ \dfrac{x - 10}{5}, & 10 < x < 15, \\ 1, & 15 \le x \le 20, \\ \dfrac{25 - x}{5}, & 20 < x < 25, \\ 0, & x \ge 25, \end{cases}$$

and

$$\mu_{\text{Large}}(x) = \begin{cases} 0, & x \le 20, \\ \dfrac{x - 20}{5}, & 20 < x < 25, \\ 1, & x \ge 25. \end{cases}$$

The overlap reflects the vagueness of the labels: a weight near 22 kg partially satisfies both `Medium` and `Large`.

Quick plotting snippet. With `scikit-fuzzy` or plain `matplotlib` you can visualize overlaps to debug label choices:

```
import numpy as np, matplotlib.pyplot as plt
x = np.linspace(0, 30, 400)
mu_small  = np.clip(1 - (x-10)/5, 0, 1) * (x <= 15)
mu_med    = np.clip((x-10)/5, 0, 1) * (x < 15)
mu_med   += ((x>=15) & (x<=20))
mu_med   += np.clip((25-x)/5, 0, 1) * (x > 20)
mu_large  = np.clip((x-20)/5, 0, 1)
plt.plot(x, mu_small, label="Small")
```

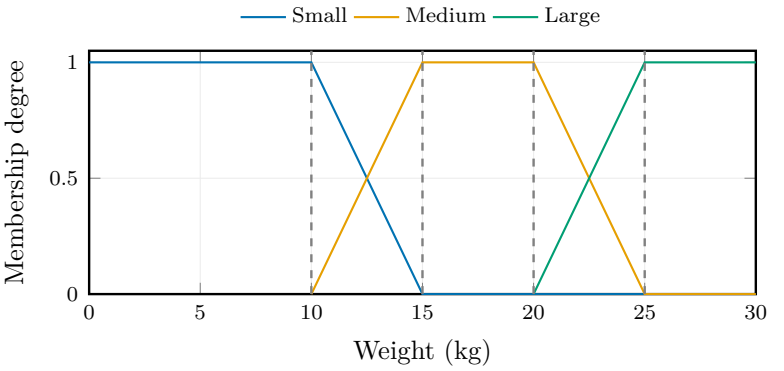

Figure 66: Schematic: Overlapping membership functions for the "Small", "Medium", and "Large" weight labels. The shaded overlaps capture gradual transitions.

```
plt.plot(x, mu_med, label="Medium")
plt.plot(x, mu_large, label="Large")
plt.legend(); plt.show()
```

16.8 Fuzzy Sets: Core Concepts and Terminology

Recall that a *fuzzy set* A on a universe X is characterized by a membership function $\mu_A : X \to [0, 1]$, where $\mu_A(x)$ quantifies the degree to which element x belongs to A. Unlike crisp sets, where membership is binary (0 or 1), fuzzy sets allow partial membership.

Support Set The *support* of a fuzzy set A is the set of all elements with nonzero membership:

$$\text{supp}(A) = \{x \in X \mid \mu_A(x) > 0\}. \tag{16.10}$$

This set captures all elements that belong to A to some degree.

Core Set The *core* of A is the set of elements fully belonging to A:

$$\text{core}(A) = \{x \in X \mid \mu_A(x) = 1\}. \tag{16.11}$$

The core set generalizes the notion of crisp membership to fuzzy sets.

Normality A fuzzy set A is said to be *normal* if there exists at least one element $x \in X$ such that $\mu_A(x) = 1$. Otherwise, A is *subnormal*. Normality ensures the fuzzy set has at least one element fully included.

Crossover Points For many membership functions, especially triangular or trapezoidal shapes, the *crossover points* c_A^- and c_A^+ are defined as the points where the membership function crosses the value 0.5:

$$\mu_A(c_A^-) = \mu_A(c_A^+) = 0.5. \tag{16.12}$$

These points are useful for interpreting the "core region" and the "fuzzy boundary" of the set.

Open and Closed Fuzzy Sets - A *left-shoulder set* reaches membership 1 for sufficiently small x and then decreases smoothly (useful for labels such as "Very Cold"). - A *right-shoulder set* mirrors this behavior for large x (e.g., "Very Hot"). - A *closed fuzzy set* has a membership function that attains 1 only on a bounded interval, typically forming a trapezoidal or triangular shape.

These distinctions help in modeling asymmetric uncertainties or preferences.

16.9 Probability vs. Possibility

It is crucial to distinguish between *probability* and *possibility* when interpreting membership functions:

- **Probability** measures the likelihood of an event occurring based on frequency or relative occurrence in repeated trials. Probabilities of mutually exclusive and exhaustive events sum to 1:

$$\sum_i P(E_i) = 1.$$

  For example, the probability that a ball drawn from a bag is red, blue, or black sums to 1.

- **Possibility**, on the other hand, measures the degree of plausibility or evidence supporting an event, without requiring additivity or summation to 1. Possibility values reflect uncertainty or vagueness rather than fre-

quency. For example, a doctor's confidence in a surgery's success might be expressed as a possibility of 0.75, indicating a degree of belief rather than a statistical frequency.

Thus, membership functions in fuzzy sets represent *possibility* rather than *probability*. This distinction is fundamental in fuzzy logic and inference (cf. Table 6 in Chapter 15). When treating a membership as a possibility distribution $\pi(x)$, we usually normalize so that $\sup_x \pi(x) = 1$, yielding $\Pi(A) = \sup_{x \in A} \pi(x)$ and $N(A) = 1 - \Pi(A^c)$.

Alpha-cuts, convexity, and fuzzy numbers

- **Alpha-cut:** $A_\alpha = \{x \in X \mid \mu_A(x) \geq \alpha\}$ for $\alpha \in (0, 1]$; A_0 is the support. Alpha-cuts turn fuzzy sets into nested crisp sets.
- **Convex fuzzy set:** A is convex if every alpha-cut A_α is convex. Normal + convex fuzzy sets with bounded support are called *fuzzy numbers*.
- **Why it matters:** Alpha-cuts commute with many continuous/monotone maps, making them a practical tool for the extension principle (Chapter 17) and for defuzzification (centroid in Chapter 18).

16.10 Fuzzy Set Operations

Fuzzy logic introduces operations on fuzzy sets that generalize classical set operations but operate on membership functions. Let A and B be fuzzy sets on X with membership functions μ_A and μ_B.

Union The union $A \cup B$ is defined by the membership function:

$$\mu_{A \cup B}(x) = \max\big(\mu_A(x), \mu_B(x)\big). \tag{16.13}$$

This generalizes the classical union by taking the maximum membership degree at each element.

Intersection The intersection $A \cap B$ is defined by:

$$\mu_{A\cap B}(x) = \min\big(\mu_A(x), \mu_B(x)\big). \tag{16.14}$$

This corresponds to the minimum membership degree, reflecting the degree to which x belongs to both sets.

Complement The complement A^c is given by:

$$\mu_{A^c}(x) = 1 - \mu_A(x). \tag{16.15}$$

This generalizes the classical complement by inverting the membership degree. Parameterized complements C_λ (e.g., Yager, Sugeno classes) are sometimes used to alter the "steepness" of the negation; they rarely satisfy involution ($C(C(x)) = x$). A common Sugeno form is

$$C_p(\mu) = \frac{1-\mu}{1+p\,\mu}, \quad p \geq 0,$$

which preserves $C_p(0) = 1$ and $C_p(1) = 0$ but is involutive only when $p = 0$. Whenever strict involution is required (as in many De Morgan identities), we default to the standard complement $C(\mu) = 1 - \mu$.

Remarks These operations satisfy properties analogous to classical set theory but adapted to fuzzy membership values. For completeness, De Morgan's laws in fuzzy logic can be written either as equivalences between sets or explicitly in max/min form:

$$\mu_{(A\cap B)^c}(x) = \mu_{A^c\cup B^c}(x) = \max\big(1-\mu_A(x),\, 1-\mu_B(x)\big), \tag{16.16}$$

$$\mu_{(A\cup B)^c}(x) = \mu_{A^c\cap B^c}(x) = \min\big(1-\mu_A(x),\, 1-\mu_B(x)\big). \tag{16.17}$$

Throughout the book we adopt $\wedge = \min$ and $\vee = \max$ as the default t-/s-norm pair with the standard complement $1 - \mu$ (the De Morgan triple used again in Chapter 18 unless noted otherwise); alternative norms appear later in operator tables.

Reminder on basic operators Equations (16.13) to (16.15) already define the max/min/standard-complement pair that we use by default. Rather than restate them, we emphasise their practical role: unions aggregate rule consequents, intersections combine antecedents, and complements capture linguistic negations. The thermostat example later in the chapter uses these defaults unless stated otherwise.

16.11 Graphical Interpretation

For continuous universes, the union and intersection membership functions can be visualized as the pointwise maximum and minimum of the two membership curves, respectively. The complement is obtained by reflecting the membership function about the horizontal line $\mu = 0.5$: every membership degree m is mapped to $1 - m$.

16.12 Additional Fuzzy Set Operations

Beyond the basic operations, several other algebraic operations are defined on fuzzy sets:

Algebraic Product The algebraic product of fuzzy sets A and B is defined by the product of their membership values:

$$\mu_{A \cdot B}(x) = \mu_A(x) \cdot \mu_B(x), \quad \forall x \in X. \tag{16.18}$$

Scalar Multiplication Given a scalar $\alpha \in [0, 1]$, scalar multiplication of a fuzzy set A is:

$$\mu_{\alpha A}(x) = \alpha \cdot \mu_A(x), \quad \forall x \in X. \tag{16.19}$$

Algebraic Sum The algebraic sum of fuzzy sets A and B is given by:

$$\mu_{A+B}(x) = \mu_A(x) + \mu_B(x) - \mu_A(x) \cdot \mu_B(x), \quad \forall x \in X. \tag{16.20}$$

This operation ensures the resulting membership values remain within $[0, 1]$.

Difference The difference between fuzzy sets A and B, denoted $A - B$, can be defined as:

$$\mu_{A-B}(x) = \mu_A(x) \wedge \big(1 - \mu_B(x)\big) = \min\big(\mu_A(x), 1 - \mu_B(x)\big), \tag{16.21}$$

where $\wedge$ denotes the minimum operator.

Bounded Difference An alternative definition of difference is the bounded difference:

$$\mu_{A \ominus B}(x) = \max\big(0, \mu_A(x) - \mu_B(x)\big). \tag{16.22}$$

Remarks:

- The difference operation in (16.21) corresponds to the intersection of A with the complement of B.
- The bounded difference in (16.22) ensures membership values remain non-negative.
- These operations extend classical set difference to fuzzy sets, but their interpretations can vary depending on the application.

16.13 Example: Union and Intersection of Fuzzy Sets

Consider two fuzzy sets

16.14 Cartesian Product of Fuzzy Sets

Recall that fuzzy sets are characterized by membership functions assigning to each element a membership grade in $[0, 1]$. When dealing with two fuzzy sets A and B defined on universes X and Y respectively, the *Cartesian product* $A \times B$ is a fuzzy relation on the product space $X \times Y$.

Definition: The membership function of the Cartesian product $A \times B$ is defined as

$$\mu_{A \times B}(x, y) = \min\big(\mu_A(x), \mu_B(y)\big), \quad \forall x \in X, y \in Y. \tag{16.23}$$

This operation generalizes the classical Cartesian product of crisp sets to fuzzy sets by taking the minimum membership grade of the paired elements.

Example: Suppose

$$A = \{(x_1, 1.0), (x_2, 0.8), (x_3, 0.4)\}, \quad B = \{(y_1, 0.6), (y_2, 0.8), (y_3, 1.0)\}.$$

Then the Cartesian product $A \times B$ is represented by the matrix of membership values:

$\mu_{A\times B}(x, y)$	y_1	y_2	y_3
x_1	$\min(1.0, 0.6) = 0.6$	$\min(1.0, 0.8) = 0.8$	$\min(1.0, 1.0) = 1.0$
x_2	$\min(0.8, 0.6) = 0.6$	$\min(0.8, 0.8) = 0.8$	$\min(0.8, 1.0) = 0.8$
x_3	$\min(0.4, 0.6) = 0.4$	$\min(0.4, 0.8) = 0.4$	$\min(0.4, 1.0) = 0.4$

Note that the Cartesian product lifts the fuzzy sets from one-dimensional membership functions to a two-dimensional fuzzy relation.

16.15 Properties of Fuzzy Set Operations

The fuzzy set operations (union, intersection, complement) satisfy several important algebraic properties analogous to classical set theory, but defined in terms of membership functions.

Commutativity:

$$\mu_{A\cap B}(x) = \mu_{B\cap A}(x), \tag{16.24}$$

$$\mu_{A\cup B}(x) = \mu_{B\cup A}(x). \tag{16.25}$$

Associativity:

$$\mu_{(A\cap B)\cap C}(x) = \mu_{A\cap(B\cap C)}(x), \tag{16.26}$$

$$\mu_{(A\cup B)\cup C}(x) = \mu_{A\cup(B\cup C)}(x). \tag{16.27}$$

Distributivity:

$$\mu_{A\cup(B\cap C)}(x) = \mu_{(A\cup B)\cap(A\cup C)}(x), \tag{16.28}$$

$$\mu_{A\cap(B\cup C)}(x) = \mu_{(A\cap B)\cup(A\cap C)}(x). \tag{16.29}$$

Identity Elements:

$$\mu_{A\cup\emptyset}(x) = \mu_A(x), \tag{16.30}$$

$$\mu_{A\cap X}(x) = \mu_A(x), \tag{16.31}$$

where $\emptyset$ is the empty fuzzy set with membership zero everywhere, and X is the universal fuzzy set with membership one everywhere.

Involution:

$$\mu_{(A^c)^c}(x) = \mu_A(x), \tag{16.32}$$

In operator notation this reads $C(C(\mu_A(x))) = \mu_A(x)$: applying the complement twice recovers the original membership degree. For the standard fuzzy complement $C(\mu_A(x)) = 1 - \mu_A(x)$, involution is just the identity

$$1 - \big(1 - \mu_A(x)\big) = \mu_A(x),$$

so the membership "returns" to its original value after two applications.

De Morgan's Laws: With the standard complement A^c and the max/min operators in Equations (16.13) to (16.15), the classical De Morgan identities hold: $(A \cap B)^c = A^c \cup B^c$ and $(A \cup B)^c = A^c \cap B^c$.

These properties ensure that fuzzy set operations behave in a consistent and algebraically sound manner, enabling the extension of classical set theory to fuzzy logic.

16.16 Fuzzy Set Operators

While operations such as union, intersection, and complement define how to combine or modify fuzzy sets, *operators* formalize the logic or rules by which these combinations occur. Operators are mappings that take one or more fuzzy sets and produce another fuzzy set, often encapsulating specific logical or algebraic behavior.

Examples of Operators:

- **Equality operator:** Checks if two fuzzy sets are equal by comparing membership functions.

16.17 Complement Operators in Fuzzy Logic

In classical logic, the complement of a proposition A is simply $1 - \mu_A(x)$, where $\mu_A(x)$ is the membership function of A. However, in fuzzy logic, this complement operation can be generalized to allow more flexible modeling of uncertainty and partial membership.

Standard Complement The standard complement operator is defined as: the standard fuzzy negation $C(\mu) = 1 - \mu$, so $\mu_{A^c}(x) = 1 - \mu_A(x)$ as in Equation (16.15). This operator is linear and intuitive but may not capture all nuances of uncertainty.

Parameterized Complement Operators To generalize the complement, choose a negation operator $C_p : [0,1] \to [0,1]$ and apply it pointwise: $\mu_{C_p(A)}(x) = C_p(\mu_A(x))$. One common (Sugeno-type) family is

$$C_p(\mu) = \frac{1-\mu}{1+p\mu}, \qquad p \geq 0, \tag{16.33}$$

which reduces to the standard complement when $p = 0$.

Another simple family is a power-law negation:

$$C_p(\mu) = (1-\mu)^p, \qquad p > 0, \tag{16.34}$$

which recovers the standard complement when $p = 1$ and adjusts the steepness for other p.

These operators allow for a nonlinear mapping of the complement, reflecting different degrees of confidence or hesitation in the membership values. Unlike the standard complement, most parameterized families *do not* preserve involution $C(C(\mu)) = \mu$ for arbitrary p; they are typically designed to satisfy boundary conditions and monotonicity instead. When strict involution is required, it is safest to use the standard complement.

Properties of Complement Operators A commonly desired set of properties for a complement operator C is:

- **Boundary conditions:** $C(0) = 1$ and $C(1) = 0$.

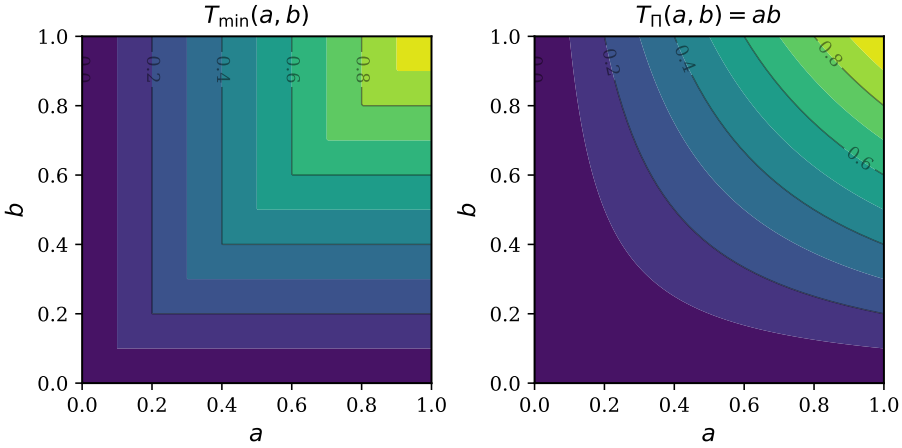

Figure 67: Schematic: Fuzzy AND surfaces comparing minimum versus product t-norms; analogous OR surfaces show similar differences. Choices here influence rule aggregation in Chapter 18.

- **Monotonicity:** $\mu_A(x) \leq \mu_B(x) \implies C(\mu_A(x)) \geq C(\mu_B(x))$.
- **Involution (optional):** $C(C(\mu_A(x))) = \mu_A(x)$.

The standard complement satisfies all three. Parameterized complements typically satisfy the first two, while involution may be relaxed to gain extra modeling flexibility; one should check involution explicitly if it is required by a particular application.

16.18 Triangular norms (t- norms)

Motivation In fuzzy logic, the logical `AND` operation is generalized by *triangular norms* (t-norms). These are binary operators that combine membership values while preserving certain desirable properties analogous to intersection in classical set theory.

Definition A **t-norm** is a binary operator $T : [0,1]^2 \to [0,1]$ satisfying the following properties for all $x, y, z \in [0,1]$:

1. **Commutativity:**

$$T(x,y) = T(y,x).$$

2. **Associativity:**
$$T(x, T(y, z)) = T(T(x, y), z).$$

3. **Monotonicity:**
$$x \leq x', \quad y \leq y' \implies T(x, y) \leq T(x', y').$$

4. **Boundary condition (Identity):**
$$T(x, 1) = x, \quad T(x, 0) = 0.$$

These properties ensure that T behaves like a generalized intersection operator.

Examples of t-norms

- **Minimum t-norm:**
$$T_{\min}(x, y) = \min(x, y).$$
This corresponds to the classical intersection in fuzzy sets.

- **Algebraic product t-norm:**
$$T_{\text{prod}}(x, y) = x \cdot y.$$
This is a smooth, multiplicative generalization of intersection.

- **Łukasiewicz t-norm:**
$$T_{\text{Luk}}(x, y) = \max(0, x + y - 1).$$

Each t-norm captures different semantics of conjunction in fuzzy logic.

Interpretation The t-norm generalizes the classical intersection operator to fuzzy sets by ensuring the output membership value remains within $[0, 1]$ and respects the ordering and boundary conditions expected of an intersection.

16.19 Triangular conorms (t- conorms / s-norms)

Definition The dual concept to t-norms is the **triangular conorm** (t-conorm), also called an *s-norm*, which generalizes the logical `OR` operation. A t-conorm $S : [0,1]^2 \to [0,1]$ satisfies:

1. **Commutativity:**
$$S(x, y) = S(y, x).$$

2. **Associativity:**
$$S(x, S(y, z)) = S(S(x, y), z).$$

3. **Monotonicity:** If $x \leq x'$ and $y \leq y'$, then
$$S(x, y) \leq S(x', y').$$

4. **Boundary conditions:**
$$S(x, 0) = x, \qquad S(x, 1) = 1.$$

These axioms mirror those of t-norms but with 1 as the neutral element instead of 0. Standard examples include the maximum s-norm $S_{\max}(x, y) = \max(x, y)$, the algebraic sum $S_{\text{sum}}(x, y) = x + y - xy$, and the bounded sum $S_{\text{bs}}(x, y) = \min(1, x + y)$; explicit formulas and their dual t-norms appear in the next subsection. Note that the algebraic sum explicitly enforces $S_{\text{sum}}(x, y) = x + y - xy \leq 1$ for all $x, y \in [0, 1]$.

16.20 T-Norms and S-Norms: Complementarity and Properties

Recall that a *t-norm* is a binary operator $T : [0,1]^2 \to [0,1]$ modeling the fuzzy intersection, and an *s-norm* (or t-conorm) $S : [0,1]^2 \to [0,1]$ models the fuzzy union. These operators satisfy certain axioms such as commutativity, associativity, monotonicity, and boundary conditions:

$$\begin{cases} T(x, 1) = x, & T(x, 0) = 0, \\ S(x, 0) = x, & S(x, 1) = 1, \end{cases}$$

for all $x \in [0, 1]$.

An important relationship between t-norms and s-norms is their complementarity via a negation operator. Throughout this section we use the *standard* fuzzy negation $N(x) = 1 - x$, so that the complement of μ_A is

$$\mu_{A^c}(x) = 1 - \mu_A(x).$$

With this explicit choice of negation, the complementarity between T and S reads:

$$T(\mu_A(x), \mu_B(x)) = 1 - S(1 - \mu_A(x), 1 - \mu_B(x)), \tag{16.35}$$

and equivalently,

$$S(\mu_A(x), \mu_B(x)) = 1 - T(1 - \mu_A(x), 1 - \mu_B(x)).$$

This duality ensures that the fuzzy intersection and union are consistent with respect to complementation, generalizing classical De Morgan's laws.

16.21 Examples of common t- norm/s-norm pairs

Several standard t-norms and their corresponding s-norms are widely used:

- **Minimum t-norm and maximum s-norm:**

$$T_{\min}(x, y) = \min(x, y), \quad S_{\max}(x, y) = \max(x, y).$$

- **Algebraic product t-norm and algebraic sum s-norm:**

$$T_{\text{prod}}(x, y) = x \cdot y, \quad S_{\text{sum}}(x, y) = x + y - xy.$$

- **Bounded difference t-norm and bounded sum s-norm:**

$$T_{\text{bd}}(x, y) = \max(0, x + y - 1), \quad S_{\text{bs}}(x, y) = \min(1, x + y).$$

Each of these pairs satisfies the complementarity relation (16.35).

16.22 Fuzzy Set Inclusion and Subset Relations

In classical set theory, $A \subseteq B$ means every element of A is also in B. For fuzzy sets, the notion of subset is generalized via membership functions.

Definition (Fuzzy Subset). A fuzzy set A is a *subset* of fuzzy set B, denoted $A \subseteq B$, if and only if

$$\mu_A(x) \leq \mu_B(x), \quad \forall x \in X,$$

where X is the universe of discourse.

If the inequality is strict for at least one x, i.e., $\mu_A(x) < \mu_B(x)$ for some x, then A is a *proper fuzzy subset* of B.

Interpretation: Since membership functions represent degrees of belonging, the subset relation is graded rather than binary. This leads naturally to the concept of *degree of inclusion*.

16.23 Degree of Inclusion

Because fuzzy membership values lie in $[0, 1]$, the subset relation can be quantified by a scalar measure indicating *how much* A is included in B.

For practical work we often use an *aggregate* measure:

$$\mathrm{incl}(A, B) = \frac{\sum_{x \in X} \min(\mu_A(x), \mu_B(x))}{\sum_{x \in X} \mu_A(x)}$$

for discrete universes (integrals for continuous, assuming finite mass). It summarizes how much of the mass of A lies inside B's support. A *pointwise* alternative relies on an implicator I and defines $\mathrm{Inc}(A, B) = \inf_x I(\mu_A(x), \mu_B(x))$ (see below); implicator-based grades avoid division by small μ_B and behave well when B has zeros. Both constructions satisfy $0 \leq \mathrm{incl}(A, B) \leq 1$, where 1 means A is fully included in B.

16.24 Set Operations and Inclusion Properties

Given fuzzy sets A, B, and C, the following properties hold for the standard t-norm and s-norm operations:

- If $A \subseteq B$, then $A \cap C \subseteq B \cap C$ and $A \cup C \subseteq B \cup C$. Explicitly,

  $$\mu_{A \cap C}(x) = \min(\mu_A(x), \mu_C(x)) \leq \min(\mu_B(x), \mu_C(x)) = \mu_{B \cap C}(x),$$

  and analogously for the union/max operator.

- If $A \subseteq B$, applying any t-norm T and its dual s-norm S preserves inclusion: $T(A, C) \subseteq T(B, C)$ and $S(A, C) \subseteq S(B, C)$. In terms of memberships,

$$\mu_{T(A,C)}(x) \leq \mu_{T(B,C)}(x) \quad \text{and} \quad \mu_{S(A,C)}(x) \leq \mu_{S(B,C)}(x), \ \forall x.$$

- Complements reverse inclusion: $A \subseteq B \Rightarrow B^c \subseteq A^c$ because complements flip the ordering of memberships. $\mu_{B^c}(x) = 1 - \mu_B(x) \leq 1 - \mu_A(x) = \mu_{A^c}(x)$.

16.25 Grades of Inclusion and Equality in Fuzzy Sets

Recall that in classical set theory, the notion of subset and equality is crisp: a set A is a subset of B if every element of A is also in B, and $A = B$ if they contain exactly the same elements. In fuzzy set theory, these notions are generalized via *grades* of inclusion and equality, which quantify the degree to which one fuzzy set is included in or equal to another.

Grade of Inclusion Given two fuzzy sets A and B defined on the universe X, with membership functions $\mu_A(x)$ and $\mu_B(x)$, respectively, the *grade of inclusion* of A in B, denoted $\text{Inc}(A, B)$, measures how much A is a subset of B.

One way to define this grade is:

$$\text{Inc}(A, B) = \inf_{x \in X} I\big(\mu_A(x), \mu_B(x)\big), \tag{16.36}$$

where I is an *implicator* function, often derived from a chosen t-norm T. A common choice is the Gödel implicator:

$$I(a, b) = \begin{cases} 1, & \text{if } a \leq b, \\ b, & \text{otherwise.} \end{cases}$$

Alternatively, if T is part of a residuated pair (T, I), one sometimes writes

$$\text{Inc}(A, B) = \inf_{x \in X} T\big(\mu_A(x), \mu_B(x)\big),$$

which should be interpreted as computing the tightest lower bound obtainable from the chosen T; this coincides with the implicator-based definition when I is

the residuum of T.

Example Suppose A and B are fuzzy sets with membership functions such that for some x we have $\mu_A(x) \leq \mu_B(x)$, and for others $\mu_A(x) > \mu_B(x)$. Using the Gödel implicator,

$$I_G(\mu_A(x), \mu_B(x)) = \begin{cases} 1, & \mu_A(x) \leq \mu_B(x), \\ \mu_B(x), & \mu_A(x) > \mu_B(x), \end{cases}$$

so the overall grade of inclusion is $\inf_{x \in X} I_G(\mu_A(x), \mu_B(x))$. This explicitly shows how the implicator returns the smaller membership where A exceeds B.

Grade of Equality Similarly, the *grade of equality* between fuzzy sets A and B, denoted $\mathrm{Eq}(A, B)$, measures how close the two sets are to being equal. It can be defined as:

$$\mathrm{Eq}(A, B) = \inf_{x \in X} J\big(\mu_A(x), \mu_B(x)\big), \tag{16.37}$$

where J is an equality function. One convenient choice is

$$J(a, b) = \begin{cases} 1, & \text{if } a = b, \\ T(a, b), & \text{otherwise,} \end{cases}$$

with T a t-norm, so that exact agreement receives unit credit while disagreements are down-weighted via T. Other smooth symmetry measures (e.g., $J(a, b) = 1 - |a - b|$) can also be used; the key requirement is that J be symmetric, bounded in $[0, 1]$, and reach 1 only when $a = b$.

This definition allows for a graded notion of equality, reflecting the fuzzy nature of the sets.

16.26 Dilation and Contraction of Fuzzy Sets

Motivation Constructing fuzzy sets with appropriate membership functions is a challenging task. Often, one starts with an initial fuzzy set A and wishes to generate related fuzzy sets that represent concepts such as "more or less A" or "somewhat A". This leads to the operations of *dilation* and *contraction* of fuzzy sets, which modify the membership function to reflect these linguistic hedges.

Definitions Given a fuzzy set A with membership function $\mu_A(x)$, we introduce two non-negative shape parameters constrained to $\alpha \geq 1$ (dilation gain) and $\beta \geq 1$ (contraction gain) so that the resulting hedges behave monotonically:

$$\text{Dilation:} \quad \mu_{A^{(d)}}(x) = \big(\mu_A(x)\big)^{1/\alpha}, \quad \alpha \geq 1, \tag{16.38}$$

$$\text{Contraction:} \quad \mu_{A^{(c)}}(x) = \big(\mu_A(x)\big)^{\beta}, \quad \beta \geq 1. \tag{16.39}$$

Using separate symbols α and β avoids the notational clash that occurs when a single parameter k is forced to satisfy both $0 < k \leq 1$ (for dilation) and $k \geq 1$ (for contraction). In some references these two operations are also called *expansion* and *narrowing*; we treat the terms as synonyms.

Note that:

- For dilation, $0 < \mu_A(x) < 1$ implies $\mu_A(x)^{1/\alpha} \geq \mu_A(x)$ when $\alpha \geq 1$, so every membership value moves closer to 1, making the fuzzy set "larger" or more inclusive. Setting $\alpha = 1$ leaves the set unchanged.
- For contraction, $0 < \mu_A(x) < 1$ implies $\mu_A(x)^{\beta} \leq \mu_A(x)$ when $\beta \geq 1$, so the membership values move toward 0, making the fuzzy set "smaller" or more restrictive. Again, $\beta = 1$ recovers the original set.

Properties

- The *core* of the fuzzy set, i.e., the elements with membership 1, remains unchanged under dilation or contraction because $1^{1/\alpha} = 1^{\beta} = 1$ for all positive α, β:

$$\mu_A(x) = 1 \implies \mu_{A^{(d)}}(x) = 1 \text{ and } \mu_{A^{(c)}}(x) = 1.$$

16.27 Closure of Membership Function Derivations

In this chapter, we finalize the discussion on how to generate new membership functions from existing ones using fuzzy set operations. Recall that membership functions represent fuzzy sets and encode the degree of membership of elements in a universe of discourse. The ability to manipulate these membership functions algebraically is fundamental to fuzzy logic and fuzzy inference systems.

16.27.1 Generating New Membership Functions via Set Operations

Given two membership functions, for example, $\mu_{\text{young}}(x)$ and $\mu_{\text{old}}(x)$, defined over the same universe X, we can construct new membership functions by applying the following operations:

Dilation (Expansion) Dilation increases the support of a fuzzy set, effectively "loosening" the membership criteria. For instance, dilating the old membership function yields a new fuzzy set more or less old:

$$\mu_{\text{more or less old}}(x) = \text{dilate}(\mu_{\text{old}}(x))$$

This operation broadens the range of x values considered "old" to some degree, reflecting linguistic vagueness.

Contraction (Narrowing) Contraction tightens the membership function, focusing on a core subset. For example, contracting $\mu_{\text{old}}(x)$ produces $\mu_{\text{too old}}(x)$:

$$\mu_{\text{too old}}(x) = \text{contract}(\mu_{\text{old}}(x))$$

This captures a stricter notion of "old."

Complement The complement of a fuzzy set reverses membership degrees:

$$\mu_{\text{not } A}(x) = 1 - \mu_A(x)$$

For example, $\mu_{\text{not young}}(x) = 1 - \mu_{\text{young}}(x)$.

Intersection The intersection of two fuzzy sets corresponds to the minimum of their membership functions:

$$\mu_{A \cap B}(x) = \min\{\mu_A(x), \mu_B(x)\}$$

This operation models the logical AND.

Union The union corresponds to the maximum:

$$\mu_{A \cup B}(x) = \max\{\mu_A(x), \mu_B(x)\}$$

16.27.2 Examples of Constructed Membership Functions

Using these operations, we can create nuanced fuzzy sets:

- **Not young and not old:**

$$\mu_{\text{not young and not old}}(x) = \min\left(1 - \mu_{\text{young}}(x),\, 1 - \mu_{\text{old}}(x)\right)$$

  This set captures individuals who are neither young nor old, representing a middle-aged group.

- **Young but not too old:** First, contract $\mu_{\text{old}}(x)$ to get $\mu_{\text{too old}}(x)$, then take its complement, and intersect with $\mu_{\text{young}}(x)$:

$$\mu_{\text{young but not too old}}(x) = \min\left(\mu_{\text{young}}(x),\, 1 - \mu_{\text{too old}}(x)\right)$$

  This set isolates those who are young but excludes those considered "too old," refining the concept of youthfulness.

- **More or less old:** Applying dilation to $\mu_{\text{old}}(x)$ expands the fuzzy set:

$$\mu_{\text{more or less old}}(x) = \text{dilate}(\mu_{\text{old}}(x))$$

Remark on Normality Note that some constructed membership functions may not be *normal*, i.e., their maximum membership degree may be less than 1. This reflects the inherent fuzziness and partial membership in linguistic concepts.

16.28 Implications for Fuzzy Inference Systems

The ability to generate new membership functions from a small set of base functions (e.g., μ_{young} and μ_{old}) is powerful. It allows us to encode complex human knowledge and linguistic nuances into fuzzy sets, which can then be used in fuzzy inference systems.

For example, consider an inference system with inputs:

$$\text{Age} \quad (x), \quad \text{Exercise Level} \quad (e)$$

Table 8: Schematic: Typical operator choices in fuzzy inference and their qualitative effects. Here the t-norm implements fuzzy AND, the s-norm implements fuzzy OR, and the implication shapes consequents.

t-norm $T(a, b)$	s-norm $S(a, b)$	Implication $\Rightarrow$	Qualitative behavior
$\min(a, b)$	$\max(a, b)$	Mamdani (clipping: $\min(\alpha, \mu_B)$)	Sharp, piecewise-linear surfaces; conservative.
$a \cdot b$	$a + b - ab$	Larsen (scaling: $\alpha\, \mu_B$)	Smoother transitions; products damp small activations.
$\max(0, a + b - 1)$	$\min(1, a + b)$	Bounded (e.g., Łukasiewicz)	Bounded sums; useful when saturation is desired.

and output:

$$\text{Health Status} \quad (h)$$

We can define membership functions for *age* (e.g., young, old) and *exercise level* (e.g., low, high), then use fuzzy operators (intersection, union, complement) to combine these inputs according to rules such as:

$$\text{IF Age is old AND Exercise is high}$$
$$\text{THEN Health is good}$$

In a Mamdani-style controller the conjunction "AND" is typically modeled by the minimum operator and the implication uses the same t-norm (i.e., the consequent is clipped at the firing strength). Other choices include using the product t-norm for conjunction, Larsen-style scaling for implication, and max for rule aggregation. Any alternative should be stated explicitly.

The next step is to formalize the *implication* and *aggregation* operators that map these fuzzy inputs to fuzzy outputs, and then perform *defuzzification* to obtain crisp outputs.

16.29 Worked Example: Mamdani Fuzzy Inference (End-to-End)

We illustrate a complete Mamdani pipeline with one antecedent (temperature) and one consequent (fan speed).

Universes and membership functions

- Temperature $T \in [0, 40]\,^\circ\mathrm{C}$ with fuzzy sets

$$\begin{aligned} \mu_{\mathrm{Cold}}(t) &= \max\Big(0,\ 1 - \tfrac{t-0}{15-0}\Big) && (0, 0, 15), \\ \mu_{\mathrm{Warm}}(t) &= \max\Big(0,\ 1 - \tfrac{|t-20|}{10}\Big) && (10, 20, 30), \\ \mu_{\mathrm{Hot}}(t) &= \max\Big(0,\ \tfrac{t-25}{40-25}\Big) && (25, 40, 40). \end{aligned}$$

- Fan speed $S \in [0, 1]$ with fuzzy sets

$$\begin{aligned} \mu_{\mathrm{Low}}(s) &= \max\Big(0,\ 1 - \tfrac{s-0}{0.5-0}\Big) && (0, 0, 0.5), \\ \mu_{\mathrm{Medium}}(s) &= \max\Big(0,\ 1 - \tfrac{|s-0.5|}{0.25}\Big) && (0.25, 0.5, 0.75), \\ \mu_{\mathrm{High}}(s) &= \max\Big(0,\ \tfrac{s-0.5}{1-0.5}\Big) && (0.5, 1, 1). \end{aligned}$$

Rule base

1. IF T is Cold THEN S is Low.
2. IF T is Warm THEN S is Medium.
3. IF T is Hot THEN S is High.

Fuzzify input and compute firing strengths For an input temperature $T = 27\,℃$,

$$\begin{aligned} \mu_{\mathrm{Cold}}(27) &= 0, \\ \mu_{\mathrm{Warm}}(27) &= \frac{30 - 27}{10} = 0.3, \\ \mu_{\mathrm{Hot}}(27) &= \frac{27 - 25}{15} = \frac{2}{15} \approx 0.133. \end{aligned}$$

Using min-implication (clipping), the consequents become

$$\begin{aligned}\mu'_{\text{Low}}(s) &= \min\big(0,\ \mu_{\text{Low}}(s)\big) = 0,\\ \mu'_{\text{Medium}}(s) &= \min\big(0.3,\ \mu_{\text{Medium}}(s)\big),\\ \mu'_{\text{High}}(s) &= \min\big(0.133,\ \mu_{\text{High}}(s)\big).\end{aligned}$$

Aggregating by max yields the overall output fuzzy set

$$\mu_{\text{out}}(s) = \max\big(\mu'_{\text{Low}}(s),\ \mu'_{\text{Medium}}(s),\ \mu'_{\text{High}}(s)\big).$$

Defuzzification (centroid) The crisp fan speed is the centroid

$$s^\star = \frac{\int_0^1 s\,\mu_{\text{out}}(s)\,ds}{\int_0^1 \mu_{\text{out}}(s)\,ds}.$$

For symmetric triangles, the centroid of a truncated Medium set remains at 0.5, and the centroid of High is at ≈ 0.833. Approximating the centroid of the max-aggregated set by a convex combination of these centroids weighted by their peak heights,

$$s^\star \approx \frac{0.3\cdot 0.5 + 0.133\cdot 0.833}{0.3 + 0.133} \approx 0.58.$$

An exact centroid can be computed analytically or numerically by integrating the clipped shapes; the approximation above matches a direct trapezoidal integration on a uniform grid (10k points), which yields $s^\star \approx 0.580$ to three decimals. See Figure 68 for the membership functions and clipping levels used in this example. Practical tip: libraries such as `scikit-fuzzy` provide a tested `defuzz` (centroid) routine; when in doubt, compute the centroid numerically rather than relying on heuristic convex combinations.

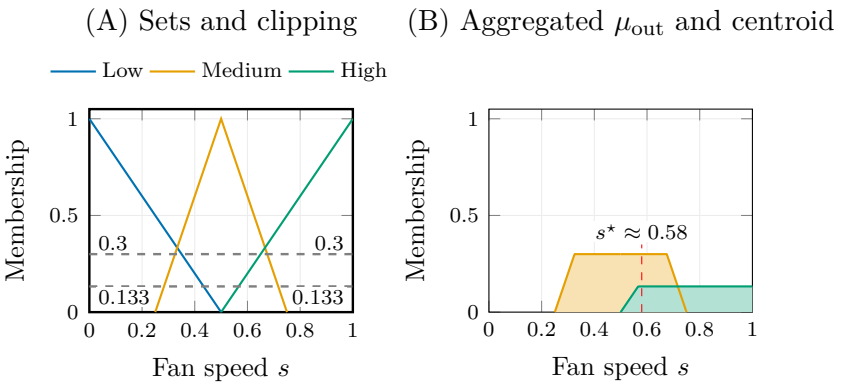

Figure 68: Schematic: End-to-end fuzzy inference example. (A) Consequent membership functions with clipping levels from firing strengths at T = 27 deg C. (B) Aggregated output set (max of truncated consequents) and a centroid marker near s* approx 0.58.

Key takeaways

- Fuzzy sets map elements to degrees in $[0, 1]$; membership shapes (triangular, trapezoidal, Gaussian) encode semantics.
- Support/core and set operations (intersection/union/complement) generalize crisp logic.
- Visualizing membership and operations clarifies design of fuzzy controllers.

Exercises and lab ideas

- Define fuzzy labels for a new universe (e.g., vehicle speed); sketch overlapping memberships and compute degrees for 3 sample points.
- Using two different t-norm/s-norm pairs, compute the union/intersection of two fuzzy sets at specific points; comment on differences.
- Write memberships for the thermostat error/rate variables (triangular or trapezoidal) and evaluate them at a few inputs.
- Plot overlapping memberships using the provided snippet; adjust parameters to see how overlap changes.

Where we head next. In Chapter 17, we move from fuzzification/defuzzification mechanics to system design patterns and adaptive fuzzy controllers; the relation operators and projections there tie these set-level tools into control and hybrid schemes.

References. Full citations for works mentioned in this chapter appear in the book-wide bibliography.

17 Fuzzy Set Transformations Between Related Universes

Learning Outcomes

- Apply the extension principle (single and multi-variable) to transport fuzzy knowledge across domains.
- Select appropriate t-norms/s-norms and understand how those choices affect projection, dilation, and composition.
- Tie these transformations to the running thermostat/autofocus example to anticipate how inference rules will behave in Chapter 18.

Operator defaults in this trilogy

Unless stated otherwise in Chapters 16 to 18, we use the standard De Morgan triple: $\wedge = \min$, $\vee = \max$, and complement $C(\mu) = 1 - \mu$. Parameterized complements (Yager/Sugeno) are noted when used, but they generally lose involution ($C(C(\mu)) \neq \mu$) unless the parameter is zero.

Running example checkpoint

We treat the thermostat's heater power as the target universe while the inputs remain error/rate. When mapping "Comfortable" from Celsius to Fahrenheit or translating error/rate pairs into control actions, the extension principle tells us how those fuzzy labels transfer; keep that single example in mind as you work through the upcoming dilation and projection formulas.

Building on Chapter 16, we address a fundamental question: *How do we*

transfer fuzzy knowledge from one universe of discourse to another related universe? This question arises whenever the same linguistic label must be reused across units, sensors, or derived variables (Celsius vs. Fahrenheit; position vs. velocity), each with its own universe and membership functions. The roadmap in Figure 1 places this in the fuzzy-relations step of the soft-computing strand.

Design motif

Preserve meaning while changing representation: the extension principle is a disciplined way to push fuzzy concepts through transformations so downstream inference remains interpretable (see Chapter 18).

17.1 Context and Motivation

Previously, we studied operations such as *dilation* and *contraction* on fuzzy sets within a single universe of discourse. For example, given a fuzzy set representing the concept *young*, we can generate related fuzzy sets like *less young* or *too old* by applying these operations. By combining these fuzzy sets, we can express nuanced concepts such as *not too young* or *not too old* within the same universe.

However, what if we want to extend this reasoning to a *different* universe of discourse that is related to the original one? For instance, consider the following scenarios:

- Mapping temperature from Celsius to Fahrenheit.
- Transforming a variable x to $y = x^2$.
- Relating speed and acceleration to derive new fuzzy sets.

In such cases, the new universe is a function of the original universe, and we want to *preserve* and *transfer* the fuzzy knowledge encoded in the original fuzzy sets to the new universe.

Notation. Throughout this chapter we use the trilogy defaults stated above: $\wedge = \min$, $\vee = \max$, and complement $1 - \mu$. When we introduce a general t-norm T, it appears explicitly (e.g., in Equation (17.2) and Equation (17.10)).

17.2 Problem Statement

Let X and Y be two universes of discourse, with a known mapping function

$$y = f(x), \quad x \in X, \quad y \in Y.$$

Suppose we have a fuzzy set $A \subseteq X$ with membership function $\mu_A : X \to [0, 1]$. We want to define a fuzzy set $B \subseteq Y$ with membership function $\mu_B : Y \to [0, 1]$ that corresponds to A under the transformation f.

The key questions are:

- How do we compute $\mu_B(y)$ for each $y \in Y$?
- How do we handle the fact that multiple $x \in X$ may map to the same $y \in Y$?
- How do we combine membership values $\mu_A(x)$ for all x such that $f(x) = y$?

17.3 Intuition and Challenges

It is tempting to define $\mu_B(y) = \mu_A(x)$ where $y = f(x)$, but this is generally insufficient because:

- The mapping f may not be one-to-one; multiple x values can map to the same y.
- Membership values represent degrees of truth or compatibility, not numerical values to be transformed arithmetically.
- Simply applying f to membership values (e.g., squaring them) does not preserve the semantic meaning of membership.

Therefore, we need a principled method to aggregate membership values from all preimages of y under f.

17.4 Formal Definition of the Transformed Membership Function

Given the fuzzy set $A \subseteq X$ with membership function μ_A, and the mapping $y = f(x)$, the membership function μ_B of the fuzzy set $B \subseteq Y$ is defined by

$$\mu_B(y) = \sup_{x \in X : f(x) = y} \mu_A(x) \tag{17.1}$$

The strongest pre-image membership determines the membership of y. When the mapping depends on multiple fuzzy variables (e.g., $f(x_1, x_2)$), the individual memberships are combined with a chosen t-norm before taking the supremum, as shown later in Equation (17.2).

Remarks:

- The sup (supremum) operator generalizes the maximum operator, capturing the highest membership value among all x mapping to y; when X is finite the supremum collapses to an ordinary maximum.
- If no $x \in X$ maps to y, then $\mu_B(y) = 0$.
- For single-input transformations no additional t-norm is needed; the aggregation shows up only when several input memberships must be combined before mapping through f.
- In continuous settings we assume f is measurable so that the pre-image sets $\{x \mid f(x) = y\}$ are well-defined and the supremum exists.

17.5 Interpretation

Equation (17.1) states that the membership degree of y in B is the supremum over all membership degrees of x in A such that $f(x) = y$. For single-input mappings no additional combination is necessary; when multiple fuzzy inputs are involved we first combine their memberships with a chosen T-norm (cf. Equation (17.2)) and then take the supremum. Intuitively, this means:

> *The degree to which y belongs to the transformed fuzzy set B is determined by the strongest membership degree among all x values that map to y, appropriately combined.*

This approach preserves the logical interpretation of membership values and respects the structure of the mapping f.

17.6 Example Setup

Consider the universe $X = \mathbb{R}$ and the fuzzy set A

17.7 Transformation of Fuzzy Sets Between Universes

We continue our discussion on fuzzy set transformations, focusing on mapping fuzzy sets from one universe to another via a function $y = f(x)$.

Example: Mapping via $y = x^2$ Consider a fuzzy set A defined on universe $X = \{-1, 0, 1, 2\}$ with membership values:

$$\mu_A(-1) = 0.340, \quad \mu_A(0) = 0.141, \quad \mu_A(1) = 0.242, \quad \mu_A(2) = 0.4.$$

Note that A is not *normal* because no element achieves membership 1; a fuzzy set is normal precisely when $\sup_{x \in X} \mu_A(x) = 1$.

Define the transformation $y = x^2$. The image universe Y consists of:

$$Y = \{0^2, (-1)^2, 1^2, 2^2\} = \{0, 1, 4\}.$$

To find the membership function $\mu_B(y)$ of the transformed fuzzy set B on Y, we use the extension principle:

$$\mu_B(y) = \sup_{x \in X : f(x) = y} \mu_A(x).$$

Calculating explicitly:

$$\begin{aligned}
&\mu_B(0) = \mu_A(0) = 0.141, \\
&\mu_B(1) = \max\{\mu_A(-1), \mu_A(1)\} = \max\{0.340, 0.242\} = 0.340, \\
&\mu_B(4) = \mu_A(2) = 0.4.
\end{aligned}$$

Thus, the transformed fuzzy set B on Y is:

$$B = \{(0, 0.141), (1, 0.340), (4, 0.4)\}.$$

Even on this very small domain the mapping $f(x) = x^2$ is *many-to-one*, because $x = -1$ and $x = 1$ both map to $y = 1$; the example therefore highlights how the supremum handles multiple pre-images.

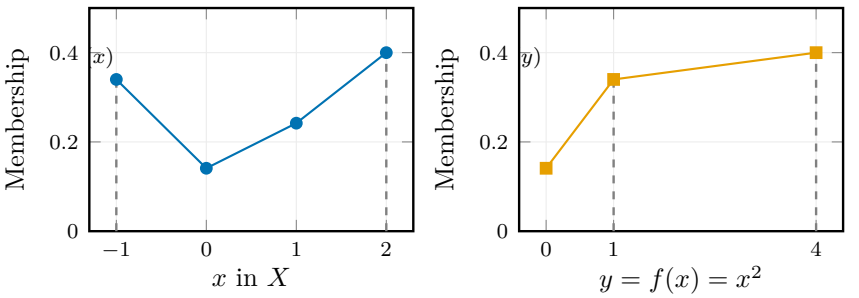

Figure 69: Schematic: Mapping a fuzzy set through the function "y = x-squared". The membership at an output value y is the supremum over all pre-images x that map to y; shared images such as x = +/-1 map to y = 1 using the maximum membership.

Worked example: monotone map (Celsius → Fahrenheit)

Let $A = \text{Comfortable}_C$ be triangular on Celsius with breakpoints $(21, 23, 25)$. For the affine map $f(x) = 1.8x + 32$, the image $B = f(A)$ is triangular with breakpoints $f(21) = 69.8$, $f(23) = 73.4$, $f(25) = 77.0$. Because f is strictly increasing, $\mu_B(y) = \mu_A(f^{-1}(y))$ and every α-cut maps directly: $B_\alpha = f(A_\alpha)$. This is the fastest way to reuse the same linguistic label across units without recomputing via (17.1).

Visual intuition. Figure 69 walks through a simple mapping $y = x^2$, showing how memberships on X lift to memberships on Y via the supremum over all pre-images that map to the same point.

Extension to Multiple Fuzzy Sets Suppose now we have two fuzzy sets A_1 and A_2 defined on the same universe $X = \{-1, 0, 1, 2\}$, with membership functions listed in the order $(-1, 0, 1, 2)$:

$$\mu_{A_1} = \{0.4, 0.7, 0.5, 0.13\}, \quad \mu_{A_2} = \{0.5, 0.1, 0.4, 0.7\}.$$

Equivalently, for A_1 we have $\mu_{A_1}(-1) = 0.4$, $\mu_{A_1}(0) = 0.7$, $\mu_{A_1}(1) = 0.5$, $\mu_{A_1}(2) = 0.13$. For A_2 we have $\mu_{A_2}(-1) = 0.5$, $\mu_{A_2}(0) = 0.1$, $\mu_{A_2}(1) = 0.4$, $\mu_{A_2}(2) = 0.7$.

Define a function $y = f(x_1, x_2) = x_1^2 + x_2^2$, where $x_1, x_2 \in X$ and their degrees

of membership are taken from A_1 and A_2 respectively.

The universe Y is the set of all possible sums of squares:

$$Y = \{x_1^2 + x_2^2 \mid x_1, x_2 \in X\}.$$

For example, some values in Y include:

$$0^2 + 0^2 = 0, \quad (-1)^2 + 0^2 = 1, \quad 1^2 + 1^2 = 2, \quad 2^2 + 2^2 = 8, \quad \ldots$$

Computing Membership Values in Y The membership function $\mu_B(y)$ is given by Zadeh's extension principle for two variables:

$$\mu_B(y) = \sup_{(x_1,x_2):f(x_1,x_2)=y} \min\{\mu_{A_1}(x_1), \mu_{A_2}(x_2)\}. \tag{17.2}$$

The minimum t-norm plays the role of the generic operator $\otimes$; any other t-norm could be substituted so long as the same choice is applied throughout the inference pipeline.

Example: Compute $\mu_B(0)$.

The pairs (x_1, x_2) such that $x_1^2 + x_2^2 = 0$ are only $(0, 0)$. Then,

$$\mu_B(0) = \min\{\mu_{A_1}(0), \mu_{A_2}(0)\} = \min\{0.7, 0.1\} = 0.1.$$

Example: Compute $\mu_B(1)$.

The pairs (x_1, x_2) such that $x_1^2 + x_2^2 = 1$ are:

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

Calculate the minimum membership values for each pair:

$$\begin{aligned}
\min\{\mu_{A_1}(-1), \mu_{A_2}(0)\} &= \min\{0.4, 0.1\} = 0.1, \\
\min\{\mu_{A_1}(0), \mu_{A_2}(-1)\} &= \min\{0.7, 0.5\} = 0.5, \\
\min\{\mu_{A_1}(1), \mu_{A_2}(0)\} &= \min\{0.5, 0.1\} = 0.1, \\
\min\{\mu_{A_1}(0), \mu_{A_2}(1)\} &= \min\{0.7, 0.4\} = 0.4.
\end{aligned}$$

Taking the supremum over all contributing pairs gives

$$\mu_B(1) = \max\{0.1, 0.5, 0.1, 0.4\} = 0.5.$$

17.8 Extension Principle Recap and Projection Operations

Recall from the previous discussion that the *extension principle* allows us to extend a fuzzy set defined on one universe to another universe via a known function. For example, if we have a fuzzy set $A \subseteq X$ and a function $f : X \to Y$, then the image fuzzy set $B = f(A) \subseteq Y$ is defined by

$$\mu_B(y) = \sup_{x \in X : f(x) = y} \mu_A(x). \tag{17.3}$$

This corresponds to taking the maximum membership value among all preimages of y under f. In discrete settings the supremum reduces to a maximum over the (finite) preimage of y; for multi-input maps $f(x_1, \ldots, x_n)$ we first combine input memberships with a chosen t-norm and then take the supremum over all tuples mapping to y.

In the continuous universe, this can become challenging because multiple x values may map to the same y, requiring careful evaluation of the supremum. The extension principle thus generalizes the image of fuzzy sets under arbitrary mappings.

Computation and discretisation tips

For discrete universes the extension principle costs $O(|X|)$ per y (or $O(|X|^n)$ for n-ary maps) because we evaluate every preimage tuple. In discrete settings sup reduces to a max. Continuous universes require discretisation: sample each input axis on a uniform or adaptive grid (typical 200–500 points per dimension), apply the t-norm/aggregation on that mesh, and approximate the supremum via max. Sparse grids or Monte Carlo sampling reduce the curse of dimensionality; always report the resolution so readers understand numeric fidelity.

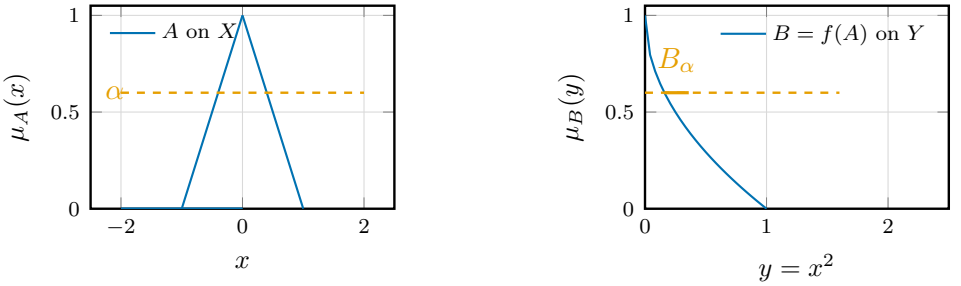

Figure 70: Schematic: Alpha-cuts under the non-monotone map "y = x-squared". A symmetric triangular fuzzy set on X maps to a right-skewed fuzzy set on Y. Each alpha-cut on A splits into two intervals whose images union to the output alpha-cut.

Alpha-cuts as an alternative

- **Unary monotone f:** $B_\alpha = f(A_\alpha)$ for every $\alpha \in (0, 1]$; computationally trivial for affine/monotone maps.
- **Non-monotone f:** split X into monotone pieces D_k; compute $B_\alpha = \bigcup_k f(A_\alpha \cap D_k)$. This is the standard route for fuzzy arithmetic on fuzzy numbers.
- **When to use:** alpha-cuts are numerically stable for continuous domains and avoid sampling artifacts when f is smooth; pointwise sup is more convenient on discrete grids.

17.9 Projection of Fuzzy Relations

Now, consider the case where we have a fuzzy relation $R \subseteq X \times Y$, where X and Y are universes of discourse. The fuzzy relation R is characterized by a membership function

$$\mu_R : X \times Y \to [0, 1].$$

This relation can be viewed as a fuzzy set on the Cartesian product $X \times Y$.

Cartesian Product of Fuzzy Sets Given fuzzy sets $A \subseteq X$ and $B \subseteq Y$ with membership functions μ_A and μ_B, their Cartesian product $R = A \times B$ is defined

by

$$\mu_R(x, y) = T(\mu_A(x), \mu_B(y)), \tag{17.4}$$

where T is a chosen t-norm, commonly the minimum operator:

$$T(a, b) = \min(a, b).$$

A *t-norm* is any binary operator $T : [0, 1]^2 \to [0, 1]$ that is commutative, associative, monotone in each argument, and has 1 as identity, so it faithfully generalizes set intersection to graded memberships. Popular choices include the minimum, the product ab, and the Łukasiewicz t-norm $\max(0, a + b - 1)$.

Table 9: Schematic: Popular t-norms and their typical roles.

T-norm	**Dual t-conorm / identity**	**When to use**
Minimum $T_{\min}(a, b) = \min(a, b)$	Dual: $\max(a, b)$; idempotent	Linguistic rules mirroring classical AND; preserves interpretability.
Product $T_{\Pi}(a, b) = ab$	Dual: probabilistic sum $a + b - ab$	Smooth gradients, probabilistic semantics, differentiable control.
Łukasiewicz $T_{\text{Luk}}(a, b) = \max(0, a + b - 1)$	Dual: bounded sum $\min(1, a + b)$	Allows partial satisfaction to accumulate; useful in preference aggregation and graded constraints; tolerates partial violations.

Example Suppose

$$\mu_A = \{0.5, 0.9\}, \quad \mu_B = \{0.8, 0.9\}.$$

Then the Cartesian product membership values are

$$\mu_R = \begin{bmatrix} \min(0.5, 0.8) & \min(0.5, 0.9) \\ \min(0.9, 0.8) & \min(0.9, 0.9) \end{bmatrix} = \begin{bmatrix} 0.5 & 0.5 \\ 0.8 & 0.9 \end{bmatrix}.$$

Here the first row corresponds to x_1, the second row to x_2, and the columns correspond to y_1 and y_2. Keeping that indexing explicit avoids ambiguity when reading off the projected membership values.

Projection of Fuzzy Relations Often, we are interested in reducing the dimensionality of a fuzzy relation by projecting it onto one of its component universes. The projection operation extracts a fuzzy set on X or Y from the fuzzy relation R.

Definition (Projection onto X). The projection of R onto X, denoted $\pi_X(R)$, is defined by

$$\mu_{\pi_X(R)}(x) = \sup_{y \in Y} \mu_R(x, y). \tag{17.5}$$

Definition (Projection onto Y). Similarly, the projection of R onto Y, denoted $\pi_Y(R)$, is defined by

$$\mu_{\pi_Y(R)}(y) = \sup_{x \in X} \mu_R(x, y). \tag{17.6}$$

Total Projection The *total projection* of R is the maximum membership value over the entire relation:

$$\mu_{\pi_{\text{total}}(R)} = \sup_{x \in X, y \in Y} \mu_R(x, y). \tag{17.7}$$

Interpretation - The projection onto X collapses the Y-dimension by taking the maximum membership value along each fixed x. - The projection onto Y collapses the X-dimension similarly. - The total projection gives the single highest membership value in the relation.

Example (continued) Using the previous example matrix for μ_R:

$$\mu_R = \begin{bmatrix} 0.5 & 0.5 \\ 0.8 & 0.9 \end{bmatrix},$$

	y_1	y_2	y_3	$\pi_X(R)$	$\pi_Y(R)$
x_1	0.9	0.3	0.1	0.9	0.9
x_2	0.4	0.8	0.2	0.8	0.8
x_3	0.1	0.6	0.5	0.6	0.5

Figure 71: Schematic: Illustrative fuzzy relation table (left) together with its projections onto the error universe (middle) and the rate-of-change universe (right). These are the exact quantities used in the running thermostat example before composing rules.

we compute

$$\mu_{\pi_X(R)} = \{\max(0.5, 0.5), \max(0.8, 0.9)\} = \{0.5, 0.9\},$$

$$\mu_{\pi_Y(R)} = \{\max(0.5, 0.8), \max(0.5, 0.9)\} = \{0.8, 0.9\},$$

and

$$\mu_{\pi_{\text{total}}(R)} = \max\{0.5, 0.8, 0.5, 0.9\} = 0.9.$$

17.10 Dimensional Extension and Projection in Fuzzy Set Operations

In practical fuzzy set operations, it is common to encounter sets defined over different universes of discourse with differing dimensions. For example, consider the union of two fuzzy sets where one is defined over a one-dimensional universe X, and the other over a two-dimensional universe $X \times Y$. To perform set operations such as union or intersection, the dimensions must be compatible.

Cylindrical Extension The *cylindrical extension* is a technique used to extend a fuzzy set defined on a lower-dimensional universe to a higher-dimensional universe by replicating membership values along the new dimension(s).

Suppose we have a fuzzy set $A \subseteq X$ with membership function $\mu_A : X \to [0, 1]$. To extend A to $X \times Y$, define the cylindrical extension A^* as:

$$\mu_{A^*}(x, y) = \mu_A(x), \quad \forall x \in X, y \in Y. \tag{17.8}$$

This operation "copies" the membership values of A uniformly along the Y-dimension, resulting in a fuzzy set over $X \times Y$.

Projection Conversely, the *projection* operation reduces the dimension of a fuzzy set by aggregating membership values over one or more dimensions. For a fuzzy set $R \subseteq X \times Y$ with membership function $\mu_R : X \times Y \to [0, 1]$, the projection onto X is again given by $\mu_{\pi_X(R)}(x) = \sup_{y \in Y} \mu_R(x, y)$ as in Equation (17.5). This operation captures the maximum membership value over all $y \in Y$ for each fixed x, effectively "collapsing" the Y-dimension.

Example Consider a fuzzy set A on $X = \{x_1, x_2\}$ with membership values $\mu_A(x_1) = 0.5$, $\mu_A(x_2) = 0.7$. Extending A cylindrically to $X \times Y$ where $Y = \{y_1, y_2, y_3\}$ yields:

$$\mu_{A^*}(x_i, y_j) = \mu_A(x_i), \quad i = 1, 2; \quad j = 1, 2, 3.$$

Thus, the membership values are replicated along the Y-axis. In practice this extension step is often paired with projections to reconcile relation dimensions before composing rules and, later, to marginalize the inferred relation back onto the universe of interest.

17.11 Fuzzy Inference via Composition of Relations

The ultimate goal of building fuzzy logic systems is to perform *inference*, i.e., to compose fuzzy rules to generate predictions or decisions. This involves combining fuzzy relations that represent knowledge or rules.

Setup Suppose we have three universes of discourse X, Y, Z, and two fuzzy relations:

$$R_1 \subseteq X \times Y, \quad R_2 \subseteq Y \times Z,$$

with membership functions $\mu_{R_1}(x, y)$ and $\mu_{R_2}(y, z)$, respectively.

The question is: can we infer a fuzzy relation $R \subseteq X \times Z$ that relates X directly to Z by composing R_1 and R_2? This is the essence of fuzzy inference.

Composition of Fuzzy Relations The composition $R = R_1 \circ R_2$ is defined by:

$$\mu_R(x, z) = \sup_{y \in Y} \min\big(\mu_{R_1}(x, y), \mu_{R_2}(y, z)\big). \tag{17.9}$$

This is known as the *sup–min composition* (or max–min composition) of fuzzy relations; replacing min with another t-norm T swaps in the chosen operator from Table 9.

Interpretation - The min operator captures the degree to which x is related to y and y is related to z simultaneously. - The sup (maximum) over all intermediate y aggregates all possible "paths" from x to z through y.

Dimensional Considerations Note that R_1 is defined on $X \times Y$, and R_2 on $Y \times Z$. The composition yields R on $X \times Z$. If the dimensions of the relations differ or if the universes are not aligned, cylindrical extension or projection can be applied to make the dimensions compatible before composition.

Example Let $X = \{x_1, x_2\}$, $Y = \{y_1, y_2\}$, and $Z = \{z_1, z_2\}$. Consider

$$\mu_{R_1} = \begin{bmatrix} 0.2 & 0.9 \\ 0.5 & 0.1 \end{bmatrix}, \qquad \mu_{R_2} = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.8 \end{bmatrix}.$$

Using the max–min composition,

$$\begin{aligned} \mu_R(x_1, z_1) &= \max\{\min(0.2, 0.7), \min(0.9, 0.4)\} = \max\{0.2, 0.4\} = 0.4, \\ \mu_R(x_1, z_2) &= \max\{\min(0.2, 0.3), \min(0.9, 0.8)\} = \max\{0.2, 0.8\} = 0.8, \\ \mu_R(x_2, z_1) &= \max\{\min(0.5, 0.7), \min(0.1, 0.4)\} = \max\{0.5, 0.1\} = 0.5, \\ \mu_R(x_2, z_2) &= \max\{\min(0.5, 0.3), \min(0.1, 0.8)\} = \max\{0.3, 0.1\} = 0.3. \end{aligned}$$

Therefore

$$\mu_R = \begin{bmatrix} 0.4 & 0.8 \\ 0.5 & 0.3 \end{bmatrix}.$$

Max–min composition as "fuzzy matrix multiply"

Given $R_1 \in [0,1]^{|X|\times|Y|}$ and $R_2 \in [0,1]^{|Y|\times|Z|}$,

```
for i in range(|X|):
    for k in range(|Z|):
        acc = 0
        for j in range(|Y|):
            acc = max(acc, min(R1[i,j], R2[j,k]))
        R[i,k] = acc
return R  # the composition R1 o R2
```

Swap min for another T (product, Łukasiewicz) and max for the corresponding s-norm to instantiate other composition families.

17.12 Recap and Motivation

Earlier in this chapter, we introduced fuzzy relations and their compositions, focusing on max–min composition as a fundamental operation. We saw how fuzzy relations can represent uncertain or imprecise mappings between sets, and how compositions allow chaining these relations to infer new relationships.

The goal of this final part is to wrap up the derivations related to fuzzy relation composition, clarify the generalization of these operations, and highlight key properties that enable their effective use in fuzzy inference systems.

17.13 Generalization of Fuzzy Relation Composition

Suppose we have two fuzzy relations:

$$R_1 \subseteq X \times Y, \quad R_2 \subseteq Y \times Z,$$

with membership functions $\mu_{R_1}(x,y)$ and $\mu_{R_2}(y,z)$, respectively.

The *composition* $R = R_1 \circ R_2$ is a fuzzy relation from X to Z defined by:

$$\mu_R(x,z) = \sup_{y\in Y} T\big(\mu_{R_1}(x,y), \mu_{R_2}(y,z)\big), \tag{17.10}$$

where T is a chosen t-norm (triangular norm) representing fuzzy conjunction (e.g., minimum, product). Recall that a t-norm $T : [0,1]^2 \to [0,1]$ is commuta-

tive, associative, monotone in each argument, and satisfies $T(a, 1) = a$; popular choices include the minimum, product, and Łukasiewicz operators.

Max–min Composition: The most common choice is the max–min composition where

$$T(a, b) = \min(a, b),$$

and the supremum is replaced by maximum:

$$\mu_R(x, z) = \max_{y \in Y} \min\left(\mu_{R_1}(x, y), \mu_{R_2}(y, z)\right).$$

This operation generalizes the classical composition of crisp relations to fuzzy sets.

17.14 Example Calculation of Composition

Consider discrete sets $X = \{x_1, x_2\}$, $Y = \{y_1, y_2\}$, and $Z = \{z_1, z_2\}$, with membership values:

$$\mu_{R_1} = \begin{bmatrix} 0.5 & 0.6 \\ 0.5 & 0.5 \end{bmatrix}, \quad \mu_{R_2} = \begin{bmatrix} 0.5 & 0.1 \\ 0.2 & 0.5 \end{bmatrix},$$

where rows correspond to X or Y elements and columns to Y or Z elements respectively.

To compute $\mu_R(x_1, z_1)$, we evaluate:

$$\mu_R(x_1, z_1) = \max\left\{\min(0.5, 0.5), \min(0.6, 0.2)\right\} = \max\{0.5, 0.2\} = 0.5.$$

Similarly, for $\mu_R(x_1, z_2)$:

$$\mu_R(x_1, z_2) = \max\left\{\min(0.5, 0.1), \min(0.6, 0.5)\right\} = \max\{0.1, 0.5\} = 0.5.$$

This process continues for all pairs (x_i, z_j) to form the composed relation matrix.

17.15 Properties of Fuzzy Relation Composition

The composition operation inherits several important algebraic properties, analogous to classical relations:

- **Associativity:** For fuzzy relations R_1, R_2, R_3,

$$(R_1 \circ R_2) \circ R_3 = R_1 \circ (R_2 \circ R_3).$$

  This allows chaining multiple relations without ambiguity.

- **Non-commutativity:** Generally,

$$R_1 \circ R_2 \neq R_2 \circ R_1,$$

  reflecting the directional nature of relations.

- **Distributivity:** Composition distributes over union:

$$R_1 \circ (R_2 \cup R_3) = (R_1 \circ R_2) \cup (R_1 \circ R_3).$$

- **De Morgan's Laws and Inclusion:** These extend naturally to fuzzy relations and their complements, intersections, and unions.

17.16 Alternative Composition Operators

While max–min is standard, other t-norms and t-conorms can be used to define composition:

- **Max-Product Composition:**

$$\mu_R(x, z) = \max_y \left(\mu_{R_1}(x, y) \cdot \mu_{R_2}(y, z) \right).$$

- **Max-Average or Other Aggregations:** Depending on application needs, different norms can be used to model conjunction and aggregation.

Author's note: choosing an operator family

Start with max–min when safety and monotonicity matter; its outputs stay within the tightest support and preserve ordering. Swap to max–product or algebraic t-norms when you need smoother surfaces or when small disagreements should be penalized multiplicatively (e.g., sensor fusion). If the resulting surfaces are too flat, tighten the t-norm; if they are too brittle, loosen it. Operator choice is an engineering dial, not an article of faith.

The choice of composition operator therefore follows a practical rule: begin with max–min for its interpretability and stability, and reach for the alternatives catalogued in Table 9 only when the application demands smoother or more aggressive aggregation.

Key takeaways

- The extension principle transfers fuzzy sets across related universes via functions $y = f(x)$.
- Multiple preimages require aggregation (e.g., sup over inverse mappings with a chosen t-norm).
- Clear notation and figures (domains, mappings) prevent ambiguity in fuzzy transformations.

Exercises and lab ideas

- Implement a minimal example from this chapter and visualize intermediate quantities (plots or diagnostics) to match the pseudocode.
- Stress-test a key hyperparameter or design choice discussed here and report the effect on validation performance or stability.
- Re-derive one core equation or update rule by hand and check it numerically against your implementation.

Where we head next. In Chapter 18, each rule induces a fuzzy relation on (input $\times$ output); sup-T composition and projection compute the implied output set before aggregation and defuzzification. That chapter uses the same defaults (max–min with the standard complement), as summarized in Equations (18.2) to (18.6).

References. Full citations for works mentioned in this chapter appear in the book-wide bibliography.

18 Fuzzy Inference Systems: Rule Composition and Output Calculation

Learning Outcomes

- Execute full Mamdani/Larsen style inference: antecedent aggregation, implication, aggregation, and defuzzification.
- Compare implication/aggregation choices (product vs. min, max vs. sum) and their impact on the running thermostat/autofocus example.
- Contrast Mamdani systems with Sugeno/Takagi–Sugeno systems to know when weighted-average consequents are preferable.

Running example checkpoint

For the thermostat, each rule combines the temperature error (*Cold*, *Slightly Warm*, …) and rate-of-change to set heater power. As you work through antecedent aggregation, implication, and defuzzification, keep one concrete rule base (e.g., "IF error is Cold AND rate is Falling THEN heater power is High") in mind; the formulas below map directly onto that setup.

Throughout this chapter we keep the trilogy defaults from Chapters 16 to 17: $\wedge = \min$, $\vee = \max$, and the standard complement $1 - \mu$. Aggregation over rule consequents defaults to the max s-norm unless stated otherwise; alternatives live in Table 9.

Building on Chapter 17, where we developed transfer operators (projection, composition, and the extension principle) to move fuzzy information between related universes, we now assemble complete fuzzy inference systems (FIS): rule composition and output calculation. The roadmap in Figure 1 shows this as the inference step that turns fuzzy sets into decisions.

Design motif

Local linguistic rules become a global behavior only after you commit to concrete operators (t-norm, implication, aggregation, defuzzifier) and then sanity-check the resulting surface.

18.1 Context and Motivation

Recall that a fuzzy inference system maps crisp inputs to fuzzy outputs by applying a set of fuzzy rules. Each rule typically has the form:

$$\textit{If } x_1 \text{ is } A_1 \textit{ and } x_2 \text{ is } A_2 \textit{ and } \cdots \textit{ then } y \text{ is } B,$$

where A_i and B are fuzzy sets defined on the respective universes of discourse. The antecedent (premise) combines multiple fuzzy conditions on inputs, and the consequent (conclusion) specifies the fuzzy output.

Author's note: rules as lived experience

These rules are not immutable physical laws; they are codified experience. We record facts such as "if it is morning then the sun is in the east," yet real observations may arrive at noon. Fuzzy inference exists to bridge that gap: observed memberships are composed with stored rules so that slight deviations in the antecedent produce softened consequents instead of brittle yes/no responses. When you carry out the algebra below, keep that picture of "experience vs. observation" in mind.

The key challenge is to systematically combine the antecedent fuzzy sets and then infer the output fuzzy set for each rule, before aggregating all rules to produce a final output.

18.2 Rule Antecedent Composition

Given a rule with n antecedents, each associated with a fuzzy set A_i and an input value x_i, the degree to which the rule is activated (also called the *firing strength*) is computed by combining the membership values of each antecedent condition.

Membership values of antecedents: For each input x_i, the membership degree in fuzzy set A_i is

$$\mu_{A_i}(x_i) \in [0, 1] \tag{18.1}$$

Aggregation operator: The combined antecedent membership is obtained by applying a fuzzy logical operator, typically the *minimum* (intersection) or the *product* operator:

$$\mu_{\text{antecedent}}(x_1, \ldots, x_n) = \min_{i=1}^{n} \mu_{A_i}(x_i), \quad \text{(min operator)} \tag{18.2}$$

$$\text{or} \quad \mu_{\text{antecedent}}(x_1, \ldots, x_n) = \prod_{i=1}^{n} \mu_{A_i}(x_i). \quad \text{(product operator)} \tag{18.3}$$

This value quantifies the degree to which the entire antecedent condition is satisfied by the input vector $\mathbf{x} = (x_1, \ldots, x_n)$. More generally, any t-norm T can be used in place of the min or product, provided it satisfies the standard properties (commutativity, associativity, monotonicity, and $T(a, 1) = a$); the chosen t-norm shapes how strictly the rule demands simultaneous satisfaction of all antecedents.

18.3 Rule Consequent and Output Fuzzy Set

Once the antecedent firing strength α is computed, it is used to modify the consequent fuzzy set B. The consequent fuzzy set is typically defined by its membership function $\mu_B(y)$ over the output universe.

Implication operator: The implication step adjusts the consequent membership function based on the firing strength α. Commonly used implication methods include:

- **Minimum implication:** Truncate the consequent membership function at level α,

$$\mu_{B'}(y) = \min\big(\alpha, \mu_B(y)\big). \tag{18.4}$$

- **Product implication:** Scale the consequent membership function by α,

$$\mu_{B'}(y) = \alpha \cdot \mu_B(y). \tag{18.5}$$

The resulting fuzzy set B' represents the *output fuzzy set* contributed by this particular rule.

Implication choices

Mamdani uses min-implication (clipping), Larsen uses product-implication (scaling). Other options include residuated and axiomatic implicators paired with their t-norms (e.g., Gödel, Product/Goguen, Łukasiewicz; see Klement et al., 2000; Dubois and Prade, 1988). Pick by desired smoothness: clipping preserves shape and interpretability; scaling yields smoother surfaces and is friendlier to gradient-based tuning.

18.4 Aggregation of Multiple Rules

When multiple rules are present, each produces an output fuzzy set B'_j with membership function $\mu_{B'_j}(y)$, where j indexes the rules. These are aggregated to form a combined output fuzzy set:

$$\mu_{B_{\text{agg}}}(y) = \max_j \mu_{B'_j}(y). \tag{18.6}$$

The *max* operator corresponds to the fuzzy union of the individual rule outputs, capturing the overall inference result.

Other aggregations Algebraic sum or bounded sum (Table 9) are used when max is too brittle; they can over-saturate when many rules fire, so start with max unless smooth blending is required.

18.5 Summary of the Fuzzy Inference Process

To summarize, the fuzzy inference process for a given input vector $\mathbf{x}$ proceeds as follows:

1. For each rule j, compute the antecedent membership degree α_j using (18.2) or (18.3).
2. Modify the consequent fuzzy set B_j by applying the implication operator (18.4) or (18.5) to obtain B'_j.
3. Aggregate all B'_j using (18.6) to obtain the overall output fuzzy set B_{agg}.

In sup-T form this is the same compositional rule of inference used in Chapter 17; here T defaults to min (or product) and sup reduces to a max on discrete grids.

The final step, defuzzification, converts B_{agg} into a crisp output value. One widely used approach is the centroid (center-of-gravity) method, which computes

$$y^* = \frac{\int_Y y\, \mu_{B_{\text{agg}}}(y)\, dy}{\int_Y \mu_{B_{\text{agg}}}(y)\, dy}. \tag{18.7}$$

This expression balances all candidate output values y by weighting them according to their membership grade in the aggregated fuzzy set. In discrete implementations, the integral is replaced with a sum over sampled output points.

Other defuzzifiers Common alternatives are mean/center of maxima (robust to multi-modal sets), smallest/largest of maxima (conservative tie-breaks), and center of sums (less sensitive to overlap than max aggregation). Choose centroid for smoothness, a max-based rule for fast or safety-critical switches, and always handle the zero-mass case explicitly.

Zero-mass fallback If the denominator in (18.7) is zero (e.g., all consequents clipped to zero), fall back to a max-membership or rule-based tie-breaker to avoid NaNs; log the condition for debugging.

Computation note With uniform sampling over m output points, centroid costs $O(mR)$ per evaluation for R rules. Non-singleton inputs add a convolution step but reuse the same aggregation/defuzz pipeline; refine the grid near peaks to reduce bias.

Centroid stability and tie-breaking

- **Multi-modal sets.** When B_{agg} has multiple peaks, the centroid may fall between modes. Log numerator and denominator separately and check that $\int \mu_{B_{\text{agg}}}(y)\, dy$ is non-zero; otherwise fall back to max-membership or a rule-based tie-break.
- **Discretisation.** Sampling the universe with too few points biases the centroid. Use uniform grids for smooth consequents and adaptive refinement near peaks for multi-modal sets. Report the step size (e.g., 0.5°C) to show numeric fidelity.

Pipeline at a glance (Mamdani/Larsen)

```
for each rule j:
    alpha_j = T( mu_A1(x1), ..., mu_An(xn) )
        # firing strength
    mu_Bj_prime(y) = implication(alpha_j, mu_Bj(y))
        # clip or scale
mu_Bagg(y) = S( mu_B1_prime(y), ..., mu_BR_prime(y) )
    # aggregate
y_star = centroid(mu_Bagg(y))
    # or another defuzzifier
```

Defaults: $T = \min$ or product; $S = \max$; centroid defuzzification. Non-singleton fuzzification (convolving input uncertainty with μ_{A_i}) uses the same pipeline once the input blend is computed.

Design checklist

- Define universes/labels; ensure coverage and reasonable overlap.
- Pick $T/S/\Rightarrow$ via Table 9 to get the smoothness/interpretability you need.
- Verify rule-base coverage; avoid contradictions/holes.
- Choose defuzzifier and sampling resolution; set a minimum-mass fallback.
- Test monotonicity/saturation; refine membership widths or rule weights.

Common pitfalls

- Max aggregation can mask contributions from several moderate rules; algebraic sum can over-saturate.
- Memberships that are too narrow yield sparse firing; too wide produce mushy outputs.
- Coarse grids bias centroids; inconsistent units across labels break interpretability.
- Neglecting non-singleton inputs: if sensor noise matters, blur inputs before fuzzifying.

18.6 Mamdani vs. Sugeno/Takagi–Sugeno systems

Mamdani-style inference (scaled fuzzy consequents, centroid defuzzification) excels when linguistic interpretability is a priority and when rule consequents must remain human-readable.
Sugeno/Takagi–Sugeno (TSK) systems replace fuzzy consequents with crisp functions such as affine models,

$$\text{IF } e \text{ is } A_i \text{ AND } \dot{e} \text{ is } B_i \text{ THEN } u_i = p_i e + q_i \dot{e} + r_i.$$

Each rule still produces a firing strength λ_i via a t-norm, but the final output becomes the weighted average

$$u^* = \frac{\sum_i \lambda_i u_i}{\sum_i \lambda_i},$$

eliminating the defuzzification integral. The trade-offs are:

- **Mamdani:** transparent consequents, straightforward incorporation of expert knowledge, but higher computational cost due to aggregation and centroid evaluation.
- **Sugeno/TSK:** faster evaluation (weighted averages), amenable to gradient-based tuning of the consequent parameters, yet less interpretable because consequents are numerical functions rather than linguistic labels.

For the thermostat example, Mamdani rules ("IF error is Cold AND rate

is Falling THEN heater power is High") are ideal when operators must audit decisions, whereas a Sugeno/TSK variant is preferable when embedding the controller into a high-speed or automatically tuned loop. [2]

Key takeaways

- Fuzzy inference composes rule antecedents (via a t-norm) and modifies consequents by implication.
- Aggregation and defuzzification (e.g., centroid) produce crisp outputs from fuzzy rule bases.
- Design choices (operators, shapes) trade interpretability and control smoothness.

Exercises and lab ideas

- Implement a Mamdani thermostat with three error labels and two rate labels; experiment with min/product t-norms and report the resulting control surfaces.
- Build a Sugeno/TSK variant of the same controller and compare outputs to the Mamdani version under identical test trajectories.
- Evaluate different defuzzification methods (centroid, weighted average, max membership) on a toy rule base and quantify the steady-state error they induce.

Where we head next. Chapter 19 moves from fuzzy controllers to evolutionary computing, where population-based search and optimization heuristics provide another pillar of soft computing.

References. Full citations for works mentioned in this chapter appear in the book-wide bibliography.

[2] Classic sources: Mamdani and Assilian (1975) for clipping implication; Takagi and Sugeno (1985) for TSK; Jang (1993) for ANFIS; see also (Klement et al., 2000; Dubois and Prade, 1988) for operator/implicator theory.

Part V: Evolutionary optimization

19 Introduction to Evolutionary Computing

Learning Outcomes

- Explain the evolutionary-computation toolbox (GAs, GP, CMA-ES, DE) and when each is well-suited.
- Implement population-based optimization loops with selection, crossover/mutation, and constraint handling.
- Diagnose convergence, premature stagnation, and feasibility trade-offs on examples such as controller tuning.

The roadmap in Chapter 1 keeps four strands in view: statistical/data-driven modeling, biological/neural representation learning, behavioral rule-based fuzzy systems, and evolutionary search. This chapter develops the evolutionary thread. See Figure 1 for the strand placement.

Design motif

When gradients are unavailable or the landscape is rugged, treat design as search: maintain a diverse population, score candidates with a fitness function, and let selection plus variation drive improvement under constraints.

19.1 Context and Motivation

Throughout this book, we have explored several complementary methodologies for building intelligent systems:

- **Statistical (data-driven) modeling:** The ERM workflow in Chapters 3 to 4 builds models by fitting objectives to datasets and auditing performance on held-out data.
- **Biological inspiration (neural networks):** The neural strand (Chapters 6 to 14) abstracts neurons and representation learning to build powerful function approximators.
- **Behavioral/rule-based reasoning (fuzzy systems):** The fuzzy trilogy

(Chapters 15 to 18) represents knowledge in linguistic terms and composes rules to act under imprecision.

Evolutionary computing adopts a distinct perspective by mimicking the process of natural evolution to tackle complex optimization problems. Readers coming directly from Chapter 18 should read this chapter as the "search" complement to the previous "rule-based" chapter: here we improve candidate solutions by selection and variation rather than by gradients or explicit rule composition.

19.2 Philosophical and Historical Background

Evolutionary computing traces its roots back to the 1950s and 1960s, contemporaneous with early developments in neural networks. It is important to recognize that evolutionary algorithms are not direct scientific models of biological evolution; rather, they are inspired by a simplified, abstracted view of evolutionary principles such as selection, mutation, and reproduction.

Key Insight: These algorithms are *heuristics*; they provide practical methods to find *good enough* solutions to problems that are otherwise computationally intractable, rather than guaranteed optimal solutions. Consequently, convergence proofs typically ensure improvement in expectation or under restrictive assumptions, but not attainment of the true global optimum.

Author's note: a pragmatic take on evolution

Evolutionary algorithms borrow the language of biology to provide a disciplined way to search rugged landscapes, not to recreate population genetics. The design mandate is pragmatic: deliver a respectable solution within the computational budget, even if it is only approximately optimal. Keep that lens in mind when evaluating selection, mutation, or recombination operators; they are tuned because they help optimization, not because they are biologically faithful.

19.3 Problem Setting: Optimization

Consider an optimization problem where the goal is to find an input vector $\mathbf{x} \in \mathbb{R}^n$ that minimizes (or maximizes) a given objective function $f : \mathbb{R}^n \to \mathbb{R}$. Formally, we want to solve

$$\mathbf{x}^* = \arg\min_{\mathbf{x}\in\mathcal{D}} f(\mathbf{x}), \tag{19.1}$$

where $\mathcal{D} \subseteq \mathbb{R}^n$ is the feasible domain incorporating any bound, equality, or inequality constraints required by the application.

Challenges:

- The function f may be *non-convex*, exhibiting multiple local minima and maxima.
- There may be no closed-form or deterministic method to find the global optimum.
- The search space $\mathcal{D}$ can be large or complex, making exhaustive search (brute force) computationally prohibitive.
- Real-time or practical constraints often require solutions within limited time frames.

19.4 Illustrative Example

Imagine a function f with multiple peaks and valleys (local maxima and minima). The global minimum is the lowest valley, but many local minima exist that can trap naive optimization methods.

Goal: Instead of guaranteeing the global optimum, evolutionary computing aims to find a *good enough* solution—one that is sufficiently close to optimal and found within a reasonable computational budget.

19.5 Why Not Brute Force?

While brute force search guarantees finding the global optimum by evaluating all possible candidates, it is often infeasible due to:

- **Computational complexity:** The number of candidate solutions can be astronomically large.
- **Time constraints:** Real-world applications often require timely decisions, making exhaustive search impractical.

For example, in control systems, one might want to tune parameters to regulate temperature or pressure optimally. Waiting for a brute force search to complete could be unacceptable, whereas a near-optimal solution found quickly is valuable.

19.6 Summary

Evolutionary computing provides a framework to address complex optimization problems by mimicking evolutionary processes. It embraces the notion of *approximate* solutions that are computationally feasible and practically useful.

The sections that follow develop the core components of evolutionary algorithms: representations, selection, crossover/mutation operators, constraint handling, and representative applications.

This "rugged landscape" picture is a useful mental model when you later interpret convergence traces, premature stagnation, and diversity metrics in population-based search.

19.7 Challenges in Continuous Optimization and Motivation for Evolutionary Computing

In many continuous optimization problems, the objective function may be undefined or discontinuous in certain regions of the domain. For example, consider a function with singularities or points where the function value is not defined (akin to division by zero). Such characteristics pose significant challenges for classical optimization methods such as gradient descent or hill climbing, which rely on smoothness and continuity to navigate the search space effectively.

Issues with Traditional Methods

- **Undefined regions:** The presence of undefined or discontinuous regions means that the gradient or directional derivatives may not exist, preventing the use of gradient-based methods.
- **Local optima and plateaus:** Even when the function is defined, it may have multiple local optima or flat regions where the gradient is zero, causing algorithms to get stuck.
- **Complex constraints:** Problems such as integer programming introduce

combinatorial constraints that are not amenable to continuous optimization techniques.

- **Computational complexity:** Many optimization problems are NP-hard, meaning no known deterministic polynomial-time algorithm can solve them exactly.

Given these challenges, deterministic approaches may be infeasible or computationally expensive. Instead, we can tolerate approximate solutions and employ heuristic or metaheuristic methods that explore the search space more flexibly. This motivates the use of *evolutionary computing* methods.

19.8 Introduction to Evolutionary Computing

Evolutionary computing (EC) is a class of algorithms inspired by the process of natural evolution. These algorithms are designed to iteratively improve candidate solutions to optimization problems by mimicking mechanisms such as selection, reproduction, and mutation observed in biological evolution.

Key Idea The goal is to design an algorithm that can solve parameter estimation or optimization problems by evolving a population of candidate solutions over successive generations. Unlike deterministic methods, evolutionary algorithms do not require gradient information or continuity and can handle complex, multimodal, and constrained problems.

Genetic Algorithms (GAs) One of the most well-known evolutionary algorithms is the Genetic Algorithm (GA). GAs attempt to *naively mimic* the process of biological evolution, albeit with a simplified and abstracted model of genetic mechanisms.

19.9 Biological Inspiration: Evolutionary Concepts

To understand GAs, we briefly review relevant biological concepts:

Chromosomes and Genes In biology, an organism's genetic information is encoded in chromosomes, which are long sequences of DNA. Each chromosome contains many genes, which determine specific traits.

Cell Division: Mitosis vs. Meiosis

- **Mitosis:** A process where a cell divides to produce two genetically identical daughter cells, each containing the full chromosome set (e.g., 46 chromosomes, i.e., 23 pairs in humans). This process is responsible for growth and tissue repair.
- **Meiosis:** A specialized form of cell division that produces gametes (sperm or egg cells) with half the number of chromosomes (haploid). When two gametes combine during fertilization, they form a new cell with a full set of chromosomes (diploid), mixing genetic material from both parents.

Genetic Recombination and Variation During meiosis, chromosomes undergo *crossover* events. Segments of genetic material are exchanged between paired chromosomes.
Recombination increases genetic diversity.

Inheritance and Heredity The offspring's chromosomes are a mixture of the parents' genetic material, but not a simple half-and-half split. Instead, genes from multiple previous generations contribute to the genetic makeup, introducing variability and enabling adaptation over time.

19.10 Implications for Genetic Algorithms

The biological processes suggest several principles that GAs incorporate:

- **Population-based search:** Maintain a population of candidate solutions (analogous to organisms).
- **Selection:** Preferentially choose better solutions to reproduce, mimicking survival of the fittest.
- **Crossover (Recombination):** Combine parts of two or more parent solutions to create offspring solutions, promoting exploration of new regions in the search space.
- **Mutation:** Introduce random changes to offspring to maintain diversity and avoid premature convergence.

- **Generational evolution:** Repeat the process over multiple generations, gradually improving solution quality.

The stochastic nature of these operations allows GAs to explore complex, multimodal landscapes and handle problems where deterministic methods struggle.

19.11 Summary of Biological Mechanisms Modeled in GAs

Biological Process	GA Analog
Chromosomes and genes	Encoding of candidate solutions (chromosomes)
Meiosis and fertilization	Crossover of parent chromosomes to produce offspring
Genetic recombination	Mixing of solution components
Mutation	Random perturbations in offspring
Selection	Fitness-based selection of parents and survivors
Generations	Iterative improvement over time

The remainder of this section formalizes how candidate solutions are encoded and how genetic operators manipulate those encodings during evolution.

GA hyperparameters at a glance

As a starting point for binary encodings of length L, choose population sizes between $5L$ and $10L$, tournament selection with small tournaments (2–4 individuals), one-point or uniform crossover, and mutation probability near $1/L$ per bit. For real-coded GAs, replace bit-flips with Gaussian mutations on parameters and use simulated binary crossover (SBX) or blend-style crossover to mix parents smoothly (Deb and Agrawal, 1995); then tune population size and mutation scale empirically based on convergence speed and population diversity.

Author's note: population and mutation heuristics

Budget dictates population size and diversity mechanisms. If evaluations are cheap, spend them on larger populations to cover the search space; if evaluations are expensive, keep populations modest but invest in diversity-preserving steps (mutation, niching, restarts) to avoid premature convergence. Use the defaults in the box above as a first pass, then tune based on convergence traces and diversity diagnostics.

19.12 Genetic Algorithms: Modeling Chromosomes

In the previous discussion, we introduced the concept of diversity in genetic algorithms (GAs) and the probabilistic nature of evolutionary processes. We now delve deeper into modeling chromosomes and the mechanisms of genetic inheritance, crossover, and mutation, drawing parallels to optimization problems.

Genetic algorithm at a glance

Objective: Optimize an objective $J(\mathbf{x})$ over a discrete or continuous search space by evolving a population of encoded candidate solutions according to selection, crossover, and mutation.
Key hyperparameters: Population size, selection pressure (tournament size or selection temperature), crossover and mutation rates, encoding scheme, and stopping criteria (max generations, no-improvement window, target fitness).
Common pitfalls: Premature convergence due to excessive selection pressure or low mutation, deceptive fitness landscapes, constraint violations when mutation or crossover produce infeasible solutions, and overinterpreting stochastic runs without multiple seeds.

Chromosomes as Information Carriers Recall that chromosomes in GAs represent candidate solutions encoded as strings of data. For modeling purposes, we consider each chromosome as a sequence of bits or symbols, each encoding a piece of information relevant to the problem domain. Formally, let a chromosome be represented as

$$\mathbf{c} = (c_1, c_2, \ldots, c_L),$$

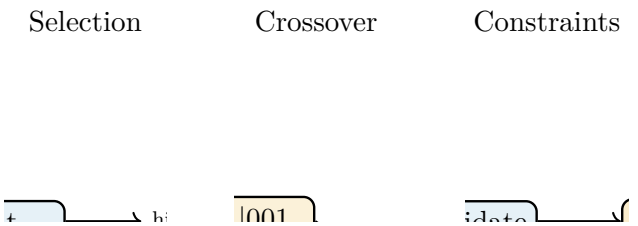

Figure 72: Schematic: Evolutionary micro-operators. Left: fitter individuals get sampled more often (roulette/tournament). Middle: crossover splices parents by a mask (one-point shown). Right: constraint handling routes offspring through repair/penalty/feasibility before evaluation.

where each gene c_i encodes a particular trait or parameter, and L is the chromosome length.

Inheritance and Crossover During reproduction, chromosomes from parent individuals combine to form offspring. This process involves:

- **Passing genes as-is:** Some genes may be inherited unchanged from a parent.
- **Crossover:** Portions of chromosomes from two parents are recombined to produce new gene sequences in offspring.
- **Mutation:** Occasionally, genes may undergo random changes, introducing new genetic material.

These mechanisms can be visualized by considering crossover as splicing two parent strings according to a binary mask and mutation as independently flipping bits with a small probability.

Modeling the Genetic Operations Let $\mathbf{p}_1$ and $\mathbf{p}_2$ be parent chromosomes. The offspring chromosome $\mathbf{o}$ is formed by combining segments from $\mathbf{p}_1$ and $\mathbf{p}_2$ according to a crossover pattern $\mathbf{x}$, and then applying mutation $\mathbf{m}$:

$$\mathbf{o} = \text{Mutate}\big(\text{Crossover}(\mathbf{p}_1, \mathbf{p}_2, \mathbf{x})\big). \tag{19.2}$$

The crossover operator selects which genes come from which parent, often

modeled by a binary mask $\mathbf{x} \in \{0,1\}^L$, where

$$o_i = \begin{cases} (p_1)_i, & \text{if } x_i = 0, \\ (p_2)_i, & \text{if } x_i = 1. \end{cases}$$

Mutation introduces random changes with a small probability μ, altering gene o_i to a different value.

Fitness and Selection Each gene or chromosome corresponds to a phenotype, representing a candidate solution with an associated fitness value $f(\mathbf{o})$. Fitness quantifies the quality or suitability of the solution, guiding the selection process for reproduction.

Consider a set of objects (e.g., facial features such as nose, eyes, lips) encoded by chromosomes. Each object variant has a fitness value reflecting its quality or adaptation. For example, fitness values might be:

$$f = \{80, 75, 60, 65, 40, 20\}$$

Over many generations, chromosomes with higher fitness values have a higher probability of surviving and reproducing, while those with lower fitness tend to be eliminated.

Probabilistic Survival and Evolution The survival probability P_s of a chromosome depends on its fitness and the evolutionary dynamics:

$$P_s(\mathbf{c}) \approx \frac{f(\mathbf{c})}{\sum_{\mathbf{c}'} f(\mathbf{c}')}.$$

Over multiple generations, this leads to the propagation of fitter chromosomes and the gradual improvement of the population.

19.13 Mapping Genetic Algorithms to Optimization Problems

Genetic algorithms can be viewed as heuristic optimization methods. To formalize this analogy, consider the components of an optimization problem:

- **Objective function:** $J(\mathbf{x})$, which we seek to maximize or minimize.

- **Constraints:** Conditions restricting the feasible set of solutions.
- **Input parameters:** Decision variables $\mathbf{x}$.

In GAs, the chromosome encodes the input parameters $\mathbf{x}$, and the fitness function corresponds to the objective function $J(\mathbf{x})$.

Key GA Components in Optimization Terms

- **Encoding:** The method of representing $\mathbf{x}$ as chromosomes.
- **Initial population:** The starting set of candidate solutions.
- **Fitness evaluation:** Computing $f(\mathbf{c}) = J(\mathbf{x})$ for each chromosome.
- **Selection:** Choosing chromosomes for reproduction based on fitness.
- **Crossover and mutation:** Generating new candidate solutions by recombining and perturbing chromosomes.
- **Convergence criteria:** Determining when the algorithm has sufficiently optimized the objective.

Constraint handling in GAs

Realistic optimization problems often involve constraints $g_j(\mathbf{x}) \leq 0$ or $h_k(\mathbf{x}) = 0$. Common strategies include (i) *repair*, which projects infeasible offspring back into the feasible set; (ii) *penalties*, which modify the fitness as $\tilde{f}(\mathbf{x}) = f(\mathbf{x}) - \rho \sum_j \max(0, g_j(\mathbf{x}))$ for some $\rho > 0$; and (iii) *feasibility rules*, which prefer feasible individuals over infeasible ones at equal objective value. Penalty methods are simple and mesh well with existing GA code, while repair and feasibility rules are preferable when constraints encode hard physical or safety limits.

Fitness as Objective Function Proxy The fitness function guides the search towards optimal solutions. The closer a chromosome's phenotype is to the desired optimum, the higher its fitness:

$$f(\mathbf{c}) \propto \text{closeness to optimum.}$$

This relationship allows GAs to explore the solution space stochastically, balancing exploitation of high-fitness regions and exploration via mutation and

19.14 Encoding in Genetic Algorithms

Recall that encoding is the process of representing the parameters of an optimization problem as a genotype, typically a string of symbols (often binary digits), which can be manipulated by genetic operators such as crossover and mutation.

Genotype and Phenotype

- **Genotype:** The encoded representation of a solution, e.g., a binary string.
- **Phenotype:** The decoded solution in the problem domain, e.g., real-valued parameters.

The goal is to design an encoding scheme that allows efficient exploration of the search space while respecting constraints and enabling effective genetic operations.

19.14.1 Common Encoding Schemes

1. Binary Encoding Each parameter is represented as a binary string of fixed length. For example, if a parameter x_i is to be represented with precision p, the length of the binary string is chosen accordingly.

- Advantages: Simple, well-studied, easy to implement crossover and mutation.
- Disadvantages: May suffer from Hamming cliffs (large phenotypic changes from small genotypic changes).

2. Floating-Point Encoding Parameters are represented directly as floating-point numbers.

- Advantages: No decoding needed, natural representation for real-valued parameters.
- Genetic operators can be adapted, e.g., crossover by averaging.
- Disadvantages: More complex mutation and crossover operators; may require specialized operators to maintain diversity.

3. Gray Coding A binary encoding where consecutive numbers differ by only one bit, reducing the Hamming distance between adjacent values.

- Useful to reduce large jumps in phenotype space due to small genotypic changes.
- Decoding involves mapping Gray code to decimal values.

19.14.2 Example: Binary Encoding of Parameters

Suppose we want to encode four parameters x_1, x_2, x_3, x_4 each represented by a binary string of length l_i. The genotype is the concatenation of these binary strings:

$$\underbrace{b_{1,1}b_{1,2}\cdots b_{1,l_1}}_{x_1} \quad \underbrace{b_{2,1}b_{2,2}\cdots b_{2,l_2}}_{x_2} \quad \underbrace{b_{3,1}b_{3,2}\cdots b_{3,l_3}}_{x_3} \quad \underbrace{b_{4,1}b_{4,2}\cdots b_{4,l_4}}_{x_4}$$

For example, a genotype might look like:

$$011 \quad 00100 \quad 0101 \quad 011110$$

Each substring is decoded to a decimal or real value according to the encoding scheme.

19.14.3 Example Problem: Minimization with Constraints

Consider the problem:

$$\min_x \quad f(x) = \frac{x}{2} + \frac{125}{x}$$

subject to

$$0 < x \leq 15$$

Encoding Strategy

- Since x is bounded between 0 and 15, we can encode x as a binary string representing integers in $[1, 15]$.
- For example, 4 bits can represent integers from 0 to 15, so we can use 4 bits and exclude zero.
- Each genotype corresponds to a candidate solution x.

Decoding

$$x = \text{decimal value of binary string}$$

If the decoded value is zero, it is invalid due to division by zero, so such genotypes are discarded or penalized.

Fitness Evaluation The fitness function corresponds to the objective function $f(x)$, possibly with penalties for constraint violations.

19.15 Population Initialization and Size

Once encoding is decided, the initial population is generated by randomly sampling genotypes within the feasible space.

Population Size

- Larger populations provide better coverage of the search space but increase computational cost.
- Smaller populations may converge prematurely.
- Typical sizes range from 20 to several hundreds depending on problem complexity.

Example For the problem above, a population of 50 individuals with 4-bit genotypes representing $x \in [1, 15]$ can be initialized by randomly generating 50 binary strings of length 4.

19.16 Genetic Operators

After initialization, genetic operators are applied to evolve the population.

19.16.1 Selection

Selection chooses individuals for reproduction based on fitness.

Common Methods

- **Roulette Wheel Selection:** Probability proportional to fitness.
- **Tournament Selection:** Randomly select a group and choose the best.

- **Rank Selection:** Rank individuals and assign selection probabilities accordingly.

19.16.2 Crossover

Crossover combines parts of two parent genotypes to produce offspring.

Binary Crossover

- **Single-point:** Choose a crossover point and swap the tail segments of two parents.
- **Two-point:** Choose two crossover points and exchange the intermediate segment.
- **Uniform:** Swap genes independently with a fixed probability.

19.17 Selection in Genetic Algorithms

After encoding candidate solutions as chromosomes (e.g., binary strings), the next step in a genetic algorithm (GA) is *selection*, which determines which chromosomes will be chosen to reproduce and form the next generation. The goal is to favor chromosomes with higher fitness, thereby guiding the search toward better solutions.

19.17.1 Fitness and Selection Probability

Given a population of N chromosomes, each chromosome i has an associated fitness value f_i. The fitness function quantifies the quality of the solution represented by the chromosome.

A common approach to selection is to assign each chromosome a probability of being chosen proportional to its fitness. This can be expressed as:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}, \quad i = 1, 2, \ldots, N, \tag{19.3}$$

where p_i is the probability that chromosome i is selected.

Roulette Wheel Selection This proportional selection method is often called *roulette wheel selection*. Imagine a wheel divided into N slices, each slice corresponding to a chromosome and sized proportionally to p_i. To select

a chromosome, a random number is generated to "spin" the wheel, and the chromosome corresponding to the slice where the wheel stops is chosen.

Key properties:

- Chromosomes with higher fitness have a larger slice and thus a higher chance of being selected.
- The same chromosome can be selected multiple times, reflecting its relative superiority.
- This stochastic process maintains diversity but can be sensitive to fitness scaling.

Example Suppose we have 5 chromosomes with fitness values:

$$f = [10, 20, 5, 15, 50].$$

The total fitness is 100, so the selection probabilities are:

$$p = [0.10, 0.20, 0.05, 0.15, 0.50].$$

Chromosome 5 has a 50% chance of selection, making it likely to be chosen multiple times.

19.17.2 Ranking Selection

When fitness values are close or vary widely, roulette wheel selection may not perform well. For example, if fitness values are very close, selection probabilities become nearly uniform, reducing selection pressure. Conversely, if one chromosome dominates, diversity may be lost prematurely.

Ranking selection addresses this by assigning selection probabilities based on the rank of chromosomes rather than raw fitness values.

Procedure

1. Sort chromosomes by fitness in descending order.
2. Assign ranks r_i such that the best chromosome has rank 1, the second best rank 2, and so forth.
3. Define a selection probability function $p(r_i)$ decreasing with rank.

A simple linear ranking scheme is:

$$p(r_i) = \frac{2-s}{N} + \frac{2(r_i-1)(s-1)}{N(N-1)}, \tag{19.4}$$

where $s \in [1, 2]$ controls selection pressure. When $s = 1$, all chromosomes have equal probability; when $s = 2$, the best chromosome has twice the average probability.

Elitism Ranking selection can be combined with *elitism*, where the best chromosome(s) are guaranteed to survive to the next generation. This ensures that the highest-quality solutions are preserved.

Advantages

- Controls selection pressure explicitly.
- Prevents premature convergence by maintaining diversity.
- Avoids issues with scaling fitness values.

Selection pressure and exploration/exploitation

- **Knobs:** Tournament size, rank-selection parameter s, crossover/mutation rates, and elitism all modulate pressure.
- **High pressure** (large tournaments, strong elitism, low mutation) speeds exploitation but risks premature convergence.
- **Low pressure** (small tournaments, higher mutation) preserves diversity but slows progress.
- **Practical default:** start with tournament size 2–3, modest elitism (top 1–5%), $p_c \approx 0.8$–0.9, and mutation tuned so 1–5% of bits/genes change per generation.

19.18 Crossover Operator

After selection, the *crossover* operator generates new offspring chromosomes by recombining parts of two parent chromosomes. This mimics biological reproduction and promotes exploration of the solution space.

19.18.1 One-Point Crossover

Consider two parent chromosomes represented as binary strings of length L:

$$\text{Parent 1: } \mathbf{c}^{(1)} = (c_1^{(1)}, c_2^{(1)}, \ldots, c_L^{(1)})$$

$$\text{Parent 2: } \mathbf{c}^{(2)} = (c_1^{(2)}, c_2^{(2)}, \ldots, c_L^{(2)})$$

One-point crossover proceeds as follows:

1. Choose a crossover point k uniformly at random from $\{1, 2, \ldots, L-1\}$.
2. Create two offspring by exchanging the tails of the parents at point k:

$$\text{Offspring 1} = (c_1^{(1)}, \ldots, c_k^{(1)}, c_{k+1}^{(2)}, \ldots, c_L^{(2)}),$$
$$\text{Offspring 2} = (c_1^{(2)}, \ldots, c_k^{(2)}, c_{k+1}^{(1)}, \ldots, c_L^{(1)}).$$

This operator allows mixing of genetic

19.19 Crossover Operators in Genetic Algorithms

In genetic algorithms, *crossover* is a fundamental operator used to combine the genetic information of two parent chromosomes to produce new offspring. The intuition behind crossover is to exchange segments of parent chromosomes to explore new regions of the solution space.

Single-point crossover is the simplest form of crossover. Given two parent chromosomes, a crossover point is selected randomly along their length. The offspring are created by taking the segment before the crossover point from the first parent and the segment after the crossover point from the second parent, and vice versa. Formally, if the parents are represented as sequences:

$$P_1 = (p_1^{(1)}, p_2^{(1)}, \ldots, p_c^{(1)}, \ldots, p_n^{(1)}), \quad P_2 = (p_1^{(2)}, p_2^{(2)}, \ldots, p_c^{(2)}, \ldots, p_n^{(2)}),$$

where c is the crossover point, then the offspring are:

$$O_1 = (p_1^{(1)}, p_2^{(1)}, \ldots, p_c^{(1)}, p_{c+1}^{(2)}, \ldots, p_n^{(2)}),$$

$$O_2 = (p_1^{(2)}, p_2^{(2)}, \ldots, p_c^{(2)}, p_{c+1}^{(1)}, \ldots, p_n^{(1)}).$$

Multi-point crossover generalizes this idea by selecting multiple crossover points. For example, in two-point crossover, two points c_1 and c_2 are chosen, and segments between these points are swapped between parents. This can be visualized as:

$$P_1 = \underbrace{p_1^{(1)} \dots p_{c_1}^{(1)}}_{\text{segment 1}} \underbrace{p_{c_1+1}^{(1)} \dots p_{c_2}^{(1)}}_{\text{segment 2}} \underbrace{p_{c_2+1}^{(1)} \dots p_n^{(1)}}_{\text{segment 3}},$$

$$P_2 = \underbrace{p_1^{(2)} \dots p_{c_1}^{(2)}}_{\text{segment 1}} \underbrace{p_{c_1+1}^{(2)} \dots p_{c_2}^{(2)}}_{\text{segment 2}} \underbrace{p_{c_2+1}^{(2)} \dots p_n^{(2)}}_{\text{segment 3}}.$$

Offspring can be generated by swapping segment 2 between parents, or by other combinations, leading to multiple possible crossover outcomes.

Probabilistic nature of crossover requires assigning a *crossover probability* p_c, which governs how often crossover is applied during reproduction. Typically, p_c is set between 0.6 and 0.9, balancing exploration and exploitation.

19.20 Mutation Operator

Mutation introduces random alterations to individual chromosomes, mimicking biological mutations. It serves to maintain genetic diversity within the population and helps the algorithm escape local optima.

Biological motivation Mutation is a rare event in nature but crucial for evolution. For example, the white coloration of polar bears is a mutation that provided an adaptive advantage in snowy environments. Similarly, environmental pressures can select for mutations, such as female elephants in Africa evolving to lack ivory tusks to avoid poaching.

Role in optimization Mutation allows the algorithm to explore new regions of the search space that are not reachable by crossover alone. Consider a fitness landscape with multiple local maxima and minima. Mutation can randomly perturb a solution, potentially moving it from a local minimum to a region near a global maximum.

Implementation of mutation In binary-encoded chromosomes, mutation typically involves flipping a bit:

$$0 \to 1, \quad 1 \to 0$$

The mutation is applied with a small *mutation probability* p_m, often on the order of 10^{-3} to 10^{-1}.

Mutation operator formalization Given a chromosome $\mathbf{x} \in \{0,1\}^n$, mutation produces $\mathbf{x}'$ by mutating each gene x_i independently with probability p_m:

$$x_i' = \begin{cases} 1 - x_i, & \text{with probability } p_m, \\ x_i, & \text{with probability } 1 - p_m \end{cases}$$

19.21 Summary of Genetic Operators and Their Probabilities

The three main genetic operators in a genetic algorithm are:

- **Selection:** Chooses chromosomes for reproduction based on fitness.
- **Crossover:** Combines genetic material from two parents to produce offspring.
- **Mutation:** Introduces random changes to chromosomes to maintain diversity.

Each operator is governed by a probability parameter:

$$\begin{aligned} p_s &= \text{probability of selection}, \\ p_c &= \text{probability of crossover}, \\ p_m &= \text{probability of mutation} \end{aligned}$$

Tuning these probabilities is critical for balancing exploration and exploitation and for avoiding premature convergence.

19.22 Known Issues in Genetic Algorithms

While genetic algorithms (GAs) provide a powerful heuristic framework for optimization, several well-known issues can affect their performance and reliability:

Premature Convergence Because GAs rely on heuristic search without a global optimality guarantee, they often converge prematurely to local minima rather than the global minimum. This is especially common if the initial population is not diverse or if the selection pressure is too high, causing loss of genetic diversity early on.

Diversity maintenance

- **Crowding/sharing:** penalize overly similar individuals to keep multiple niches in multi-modal landscapes.
- **Restarts/islands:** run multiple subpopulations (often in parallel) with occasional migration; robust against stagnation.
- **Adaptive mutation:** increase mutation or inject random individuals when diversity drops.

Mutation Interference Mutation is intended to introduce diversity and help escape local minima by randomly altering genes. However, excessive or poorly controlled mutation can cause oscillations, where beneficial mutations are undone by subsequent mutations. This back-and-forth effect can prevent convergence and degrade solution quality.

Deception Deception refers to situations where the encoding or representation of solutions misleads the GA's fitness evaluation. Low-order schemata with high observed fitness may actually guide the search away from the global optimum, so that combining "good" building blocks produces worse offspring. There is no single formal definition, but a deceptive fitness landscape is one in which local improvements inferred from schemata systematically lead the GA to suboptimal basins of attraction.

Fitness Misinterpretation Since selection is driven by fitness values, any inaccuracies or misleading fitness evaluations can cause the GA to make poor decisions about which individuals to propagate. This can arise from noisy fitness functions, poorly designed objective functions, or deceptive encodings.

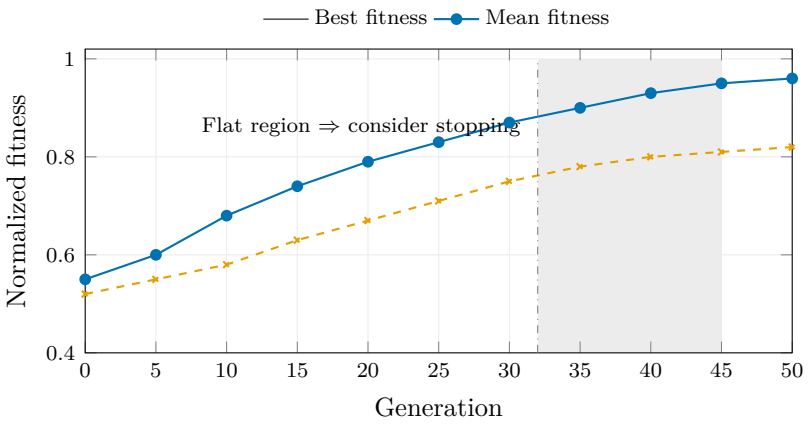

Figure 73: Schematic: Illustrative GA run showing the best and mean normalized fitness over 50 generations. Flat regions motivate "no improvement" stopping rules, while steady separation between best and mean indicates ongoing selection pressure.

19.23 Convergence Criteria

Determining when to stop the GA is a critical practical consideration. Common convergence criteria include:

- **Fixed number of generations:** Run the GA for a predetermined number of iterations.
- **Time limit:** Stop after a fixed amount of computational time.
- **No improvement:** Terminate if the best fitness value has not improved over a specified number of generations.
- **Manual inspection:** Periodically inspect the population to decide if the solutions are satisfactory.

In practice, a combination of these criteria is often used. For example, one might stop if *either* (a) no improvement in the best fitness is observed for 10 consecutive generations, *or* (b) the run reaches 100 generations in total, whichever condition is met first. Figure 73 visualizes such a run, making it easy to spot plateaus and the persistent gap between best and mean fitness.

19.24 Summary of Genetic Algorithm Workflow

To summarize the GA process:

1. **Initialization:** Generate an initial population of chromosomes (candidate solutions).
2. **Fitness Evaluation:** Compute the fitness value for each chromosome based on the objective function.
3. **Termination Check:** If a satisfactory solution is found or a stopping criterion is met, terminate.
4. **Selection:** Select parent chromosomes based on fitness (e.g., roulette wheel, tournament).
5. **Crossover:** Apply crossover operators to parents to produce offspring.
6. **Mutation:** Apply mutation operators to offspring to maintain diversity.
7. **Replacement:** Form the new population from offspring (and possibly some parents).
8. **Repeat:** Return to step 2.

This iterative cycle continues until convergence or stopping criteria are met. The flow diagram in Figure 74 summarizes the control structure on a single page for quick reference.

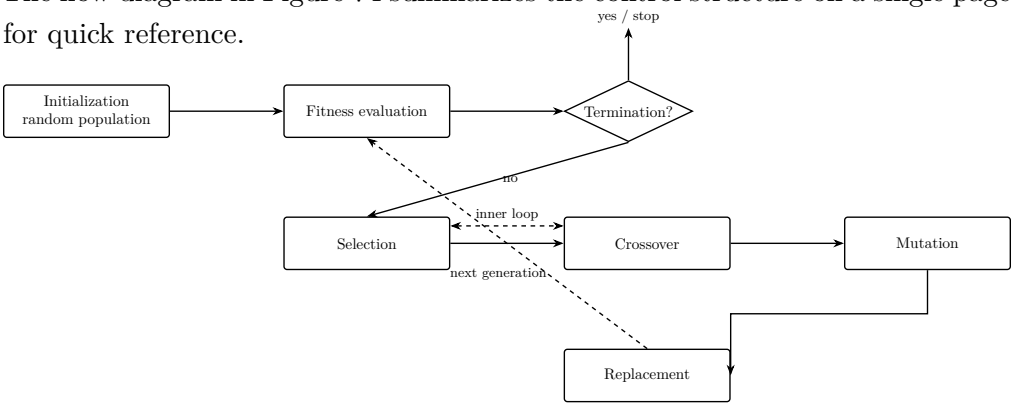

Figure 74: Schematic: GA flowchart showing the iterative process: initialization leads to fitness evaluation and a termination check. If not terminated, the algorithm proceeds through selection, crossover, mutation, and replacement, which then feeds the next generation's fitness evaluation.

Step	Example bitstrings	Decoded x	Fitness $f(x)$
Initial (subset)	0001_2, 0010_2, 0101_2, 0111_2	0.0625, 0.125, 0.3125, 0.4375	0.40, 0.13, -0.52, -0.02
Select parents (example)	0010_2, 0101_2	0.125, 0.3125	0.13, -0.52
Crossover (one-point)	Parents: 00|10, 01|01 $\rightarrow$ offspring: 00|01, 01|10	0.0625, 0.3750	0.40, 0.29
Mutation (flip one bit)	$0110_2 \rightarrow 0111_2$	0.4375	-0.02

Table 10: Schematic: Toy GA generation on a bounded interval. One crossover and mutation illustrate how the fitness function guides selection before the next generation.

Defaults that work (DE/CMA-ES)

Differential Evolution (DE): start with population $10-20 \times D$ (dimension D), mutation scale $F \in [0.5, 0.8]$, crossover $C_r \in [0.7, 0.9]$. **CMA-ES:** population $\lambda \approx 4 + \lfloor 3 \ln D \rfloor$, initial step-size $\sigma_0 \approx 0.3$ of the variable range. These defaults are robust first tries before tuning.

19.25 Pseudocode Representation

The GA can be expressed in pseudocode as follows:

```
Initialize population P with N chromosomes
Evaluate fitness of each chromosome in P

while termination criteria not met do
    Select parents from P based on fitness
    Apply crossover to parents to create offspring
    Apply mutation to offspring
    Evaluate fitness of offspring
    Replace some or all of P with offspring
end while

Return best chromosome found
```

19.26 Example: GA for a Constrained Optimization Problem

Consider the problem of minimizing the function:

$$f(x) = \cos(5\pi x) \cdot \exp\left(-x^2\right)$$

subject to the constraint:

$$0 \leq x \leq 0.5$$

with a precision of three decimal places.

GA Parameters:

- Population size: 10 chromosomes
- Encoding: Real-valued x encoded with 3 decimal places (i.e., precision of 0.001)
- Crossover probability: 25%
- Mutation probability: 10%
- Selection: All chromosomes are selected for crossover or mutation (no explicit selection probability)

Initialization: Generate 10 random values of x uniformly distributed in $[0, 0.5]$, each rounded to three decimal places. When prior designs or surrogate models exist, *warm start* a few chromosomes with those known-good solutions before filling the rest randomly; seeding accelerates convergence without losing diversity if you keep most of the population stochastic.

Fitness Evaluation: Calculate $f(x)$ for each chromosome. Since this is a minimization problem, fitness can be defined as the negative of the function value or by using a suitable transformation to ensure higher fitness corresponds to better solutions.

Evolutionary Cycle: Apply selection, crossover, and mutation to produce new offspring, then evaluate their fitness. Repeat this process for multiple generations until convergence criteria are met.

Remarks: In practice, some initial chromosomes may fall outside the constraint bounds due to rounding or mutation; these should be clipped or repaired to maintain feasibility.

As an illustration, consider an initial population with fitness values computed directly from $f(x)$:

$$\{0.041,\ 0.178,\ 0.203,\ 0.247,\ 0.311,\ 0.328,\ 0.359,\ 0.402,\ 0.435,\ 0.496\}.$$

Each chromosome uses a 9-bit fixed-point code (3 fractional digits), decoded by interpreting the bits as an integer n and scaling via $x = n/1000$. Chromosomes decoding to $x = 0$ are discarded or repaired.

A single generation could proceed as follows:

- Select the top five chromosomes by fitness.
- Apply one-point crossover at the 5th bit to produce offspring (e.g., parents 0.203 and 0.359 yield 0.209 and 0.353).
- Mutate each bit with probability 0.1, ensuring all decoded values remain within [0, 0.5].
- Re-evaluate fitness and retain the best ten individuals for the next generation.

Constraint handling playbook

- **Penalty methods** soften constraints by augmenting the fitness with a violation term, e.g., $F(\mathbf{x}) = f(\mathbf{x}) + \lambda \sum_k \max\{0, g_k(\mathbf{x})\}^2$; increase λ when infeasible individuals survive too often.
- **Repair operators** project infeasible chromosomes back into the feasible region (clip bound violations, renormalize equality constraints, or rerun a problem-specific solver) before evaluation.
- **Feasibility-first selection** ranks feasible candidates ahead of infeasible ones, then compares raw fitness only within each group; among infeasible solutions, select those with the smallest violation.
- **Deb's feasibility rules** (Deb, 2001): (i) if one solution is feasible and the other is not, pick the feasible one; (ii) if both are feasible, pick the better objective; (iii) if both are infeasible, pick the one with smaller total constraint violation. Adaptive penalties can be layered on top when violations persist.

Reproducibility and fair comparison Fix random seeds when debugging, run many seeds (e.g., 20+) for reporting, match evaluation budgets across algorithms, and report mean/median best-so-far with variability bands. Log all hyperparameters and share code/configs to make comparisons fair. Penalty terms are easy to implement, repair operators exploit domain knowledge, and feasibility-first policies are useful in safety-critical controllers where violating constraints is unacceptable even temporarily.

19.27 Genetic Algorithms: Iterative Process and Convergence

Recall that in genetic algorithms (GAs), we start with an initial population of candidate solutions, each represented by a chromosome encoding a potential solution vector. The fitness of each candidate is evaluated by plugging the encoded values into the objective function. Based on these fitness values, selection probabilities are assigned, favoring candidates with higher fitness (for maximization problems) or lower fitness (for minimization problems).

Selection and Reproduction Selection is typically stochastic but biased towards fitter individuals. For example, in a population of size $N = 10$, some individuals may be selected multiple times, while others may not be selected at all. This process ensures that better solutions have a higher chance of propagating their genetic material to the next generation.

Crossover and Mutation After selection, genetic operators such as crossover and mutation are applied:

- **Crossover:** With a certain probability (e.g., 25%), pairs of selected chromosomes exchange segments of their genetic code to produce offspring. This recombination explores new regions of the solution space by mixing existing solutions.
- **Mutation:** With a smaller probability (e.g., 10%), random changes are introduced to individual genes in the chromosomes. Mutation maintains genetic diversity and helps prevent premature convergence to local optima.

Evolution Over Generations By iterating the cycle of selection, crossover, and mutation over multiple generations, the population gradually evolves towards better solutions. Early generations may have widely dispersed candidate solutions, but as evolution proceeds, solutions cluster around local maxima or minima of the fitness landscape.

In practice, after a sufficient number of generations (e.g., 16 in the example), the algorithm often converges to a solution with the highest fitness value found so far. While this is not guaranteed to be the global optimum, it often provides a very good approximation.

19.28 Beyond canonical GAs: real-coded strategies

Bit-string encodings are ideal for combinatorial search, yet most engineering problems have continuous decision variables. Two mature real-coded families are now standard tools:

- **Covariance Matrix Adaptation Evolution Strategy (CMA-ES)** (Hansen and Ostermeier, 2001) maintains a multivariate Gaussian search distribution. Successful steps update the mean, adapt the global step size, and rotate the covariance to align with the landscape's principal directions.

CMA-ES shines on smooth, ill-conditioned black-box functions where gradients are unavailable but the objective rewards second-order adaptation.

- **Differential Evolution (DE)** (Storn and Price, 1997) perturbs a target vector with scaled differences of two other individuals, $\mathbf{v} = \mathbf{x}_r + F(\mathbf{x}_p - \mathbf{x}_q)$, then mixes $\mathbf{v}$ with the original via binomial or exponential crossover. This simple mechanism balances exploration/exploitation with only three hyperparameters (F, C_r, N) and handles noisy, non-smooth objectives well.

Both algorithms plug into the same evaluation loop shown earlier and can reuse the constraint-handling policies in the preceding box. In practice many teams prototype with DE (fast, few knobs) and switch to CMA-ES when the problem demands higher precision or adaptive covariance modeling.

19.29 Genetic Programming (GP)

Genetic programming extends the principles of genetic algorithms to the evolution of computer programs or symbolic expressions rather than fixed-length parameter vectors.

Problem Setup Consider a problem where the relationship between input variables $x_1, x_2, \ldots, x_n$ and output y is unknown. Unlike traditional parameter estimation, we do not assume a fixed functional form. Instead, we want to discover the function f such that

$$y = f(x_1, x_2, \ldots, x_n).$$

Representation of Programs In GP, candidate solutions are represented as tree-like structures encoding mathematical expressions or programs composed of:

- **Terminals:** Input variables ($x_1, x_2, \ldots$) and constants.
- **Functions:** Arithmetic operations (addition, subtraction, multiplication, division), logical operations, or other domain-specific functions.

For example, a candidate program might represent the expression

$$(x_1 \times x_3) + (x_1 + x_4).$$

Genetic Operators in GP

- **Crossover:** Subtrees from two parent programs are exchanged to create offspring programs.
- **Mutation:** Random modifications are made to nodes in the program tree, such as changing an operator or replacing a subtree.

These operations allow the evolution of increasingly complex and effective programs.

Fitness Evaluation A candidate program is evaluated by executing it on a training set and comparing its outputs with the desired targets. Fitness functions often measure mean squared error, classification accuracy, or accumulated reward, and penalize programs that raise runtime exceptions or exceed resource limits. Individuals with higher fitness are more likely to be selected for reproduction.

Example Suppose we have the following initial program trees:

$$\begin{aligned} &\text{Parent 1:} \quad f_1 = (x_1 \times x_3) + (x_1 + x_4) \\ &\text{Parent 2:} \quad f_2 = (x_2 - 5) \times (x_4 + 1) \end{aligned}$$

Suppose we exchange the right subtree of f_1 (the addition node $x_1 + x_4$) with the left subtree of f_2 (the subtraction node $x_2 - 5$). The resulting offspring are

$$f_1' = (x_1 \times x_3) + (x_2 - 5), \qquad f_2' = (x_1 + x_4) \times (x_4 + 1).$$

Mutation might then replace the terminal x_4 in f_1' with a constant (e.g., 5) or switch the addition operator to multiplication, thereby exploring nearby program structures while keeping the tree depth bounded.

Recursive and Modular Programs GP can evolve recursive functions and modular code blocks (subroutines), enabling the discovery of complex behaviors and algorithms. In practice this is achieved by allowing trees to reference automatically defined functions (ADFs) or macros that are evolved alongside the main program. The depth of the program trees and the number of reusable modules are usually constrained to prevent uncontrolled growth and to keep execution cost manageable.

Applications Genetic programming is particularly useful for:

- Symbolic regression: discovering analytical expressions fitting data.
- Automated program synthesis: generating code for control, decision-making, or data processing.
- Robotics: evolving control programs for navigation, obstacle avoidance, or manipulation.

Example: Robot Obstacle Avoidance Consider evolving a program that controls a robot's movement based on sensor inputs indicating obstacles. The function set might include commands like `move_forward`, `turn_left`, `turn_right`, and conditional statements. The GP evolves sequences and combinations of these commands to maximize the robot's ability to navigate without collisions.

Summary Genetic programming generalizes genetic

19.30 Wrapping Up Genetic Algorithms and Genetic Programming

In this final segment of the chapter, we conclude our discussion on genetic algorithms (GAs) and genetic programming (GP), emphasizing their conceptual foundations, practical implications, and the distinctions between them.

Recap of Genetic Algorithms Genetic algorithms are population-based metaheuristic optimization methods inspired by natural selection and genetics. They operate on a population of candidate solutions, iteratively applying genetic operators such as selection, crossover, and mutation to evolve solutions toward optimality. The key components include:

- **Representation:** Encoding candidate solutions as chromosomes (bit strings, real vectors, etc.).
- **Fitness Function:** Quantifies the quality of each candidate solution.
- **Genetic Operators:**
  - *Selection* favors fitter individuals.
  - *Crossover* recombines genetic material from parents.

  - *Mutation* introduces random variations.

- **Evolutionary Cycle:** Repeat selection and genetic operations until convergence or stopping criteria are met.

Genetic Programming: Structure over Parameters Genetic programming extends the GA paradigm by evolving computer programs or symbolic expressions rather than fixed-length parameter vectors. The fundamental difference is that GP searches over the space of program structures (trees of functions and terminals) instead of numeric parameter values.

Key points about GP include:

- **Representation:** Programs are represented as hierarchical trees, where internal nodes are functions (e.g., arithmetic operators, logical functions) and leaves are terminals (input variables, constants).

- **Evolution of Programs:** Genetic operators manipulate program trees:

  - *Crossover* exchanges subtrees between parent programs.

  - *Mutation* randomly modifies nodes or subtrees.

- **Fitness Evaluation:** Programs are executed on input data, and their outputs are compared against desired outputs to compute fitness.

- **Emergent Solutions:** GP can discover novel program structures that model complex phenomena without explicit programming, often yielding surprising and insightful results.

Applications and Insights Genetic programming is particularly powerful for modeling complex systems where the underlying relationships are unknown or difficult to specify explicitly. For example, given inputs such as wind speed, humidity, and temperature, GP can evolve models that predict environmental phenomena without prior assumptions about the functional form.

This capability highlights the strength of GP as a tool for automated model discovery and symbolic regression.

Further Topics and Extensions While this chapter provided a concise overview, the field of evolutionary computation encompasses many advanced

topics, including:

- **Multi-objective Genetic Algorithms:** Handling optimization problems with multiple conflicting objectives.
- **Constraint Handling:** Incorporating problem-specific constraints into the evolutionary process.
- **Hybrid Methods:** Combining GAs/GP with other optimization or machine learning techniques.
- **Scalability and Parallelization:** Efficiently implementing evolutionary algorithms for large-scale problems.

Readers are encouraged to explore these topics through further reading and research; the short primer below highlights the most widely used multi-objective GA.

19.31 Multi-objective search and NSGA-II

When two or more objectives conflict, we seek a set of Pareto-optimal solutions rather than a single best point. The Non-dominated Sorting Genetic Algorithm II (NSGA-II) sorts each generation into Pareto fronts: rank-1 individuals are non-dominated, rank-2 are dominated only by rank-1, etc. Replacement preserves all members of the best fronts and uses crowding distance to maintain diversity along the trade-off curve. NSGA-II's combination of elitist survival and $O(N \log N)$ non-dominated sorting makes it the default baseline for multi-objective evolutionary optimization (Deb et al., 2002).

Metrics and variants Hypervolume (area/volume dominated by the front with respect to a reference point) is a common scalar indicator; report it alongside the spread of solutions (Zitzler et al., 2002). MOEA/D decomposes objectives into weighted subproblems; SPEA2/IBEA are popular alternatives. Always plot the Pareto set and budget-matched hypervolume traces when comparing algorithms.

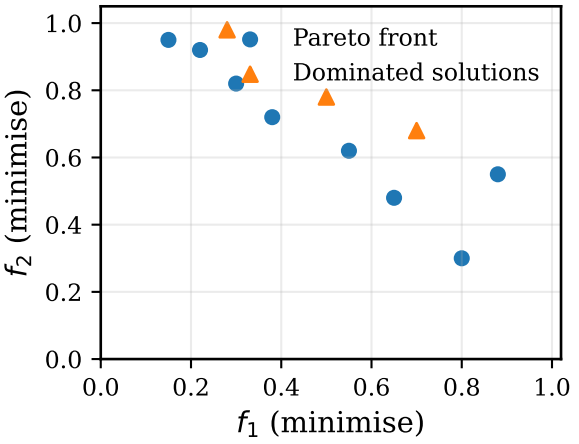

Figure 75: Schematic: Sample Pareto front for two objectives. NSGA-II keeps all non-dominated points (blue) while pushing dominated solutions (orange) toward the front via selection, yielding a spread of trade-offs in one run.

Key takeaways

- Evolutionary algorithms optimize without gradients by iterating selection, variation, and replacement under a fitness evaluation loop.
- Constraint handling is part of the design: penalties, repair, and feasibility-first selection each encode a different notion of "acceptable" search.
- Multi-objective search replaces a single optimum with a Pareto front; NSGA-II is a standard baseline for producing diverse trade-offs.

Exercises and lab ideas

- Implement a simple GA for $f(x) = \cos(5\pi x)\exp\left(-x^2\right)$, experimenting with penalty vs. repair strategies for the $[0, 0.5]$ constraint.
- Prototype a CMA-ES or Differential Evolution solver on a noisy Rosenbrock function and compare convergence traces against the canonical GA.
- Build a tiny GP to rediscover a closed-form expression (e.g., $y = x^3 + x$) from samples; report how often crossover/mutation produce valid programs.

Where we head next. This chapter closes the soft-computing thread. For background refreshers, Appendix A summarizes linear-systems prerequisites and Appendix B collects kernel-method context used earlier in the book.

References. Full citations for works mentioned in this chapter appear in the book-wide bibliography.

Key Takeaways

Chapter 1 *About This Book* explains how to read the book, points to the notation/figure conventions, and motivates the four-level taxonomy for systems thinking across chapters.

Chapter 2 *Symbolic Integration and Problem-Solving Strategies* shows how safe substitutions, heuristic branches, and numeric fallbacks cooperate in a transformation tree, giving a procedural view of algebraic problem solving.

Chapter 3 *Supervised Learning Foundations* develops ERM/MLE/MAP, contrasts loss families, and grounds diagnostics such as learning curves, calibration, and proper scoring rules.

Chapter 4 *Classification and Logistic Regression* reuses the Chapter 3 toolkit to build probabilistic classifiers, emphasize ROC/PR analysis, and reason about class imbalance, calibration, and optimization choices (Newton vs. first-order).

Chapter 5 *Introduction to Neural Networks* casts the perceptron as a thresholded linear classifier (vs. logistic as the smooth probabilistic counterpart), proves convergence guarantees, and catalogues practical pitfalls such as poor feature scaling or non-separable data.

Chapter 6 *MLP Foundations* formalizes forward/backward passes with matrix-calculus identities, highlights caching/normalization for numerical stability, and frames bias–variance behavior for deep linear stacks.

Chapter 7 *Backpropagation in Practice* turns the derivatives into SGD/mini-batch pseudocode, adds early-stopping heuristics, and compares optimization tweaks (momentum, adaptive schedules) against the diagnostics from Chapter 3.

Chapter 8 *Radial Basis Function Networks* interprets RBFs as local "bubbles," covers center/width selection (including practical σ rules), contrasts primal vs. dual training formulations, and connects the finite-basis view to kernel methods (e.g., kernel ridge regression and SVMs with RBF kernels).

Chapter 9 *Self-Organizing Maps* explains neighborhood competition/cooperation phases, quality measures (quantization/topographic error), and visualization tricks for prototype-based embedding.

Chapter 10 *Hopfield and Energy-Based Memories* derives discrete/continuous dynamics, capacity bounds, and asynchronous vs. synchronous update strategies for associative recall.

Chapter 11 *Convolutional Neural Networks and Deep Training Tools* details convolution/cross-correlation, pooling, receptive-field growth, and the engineering defaults behind modern CNN blocks and training loops.

Chapter 12 *Recurrent Neural Networks* develops BPTT, gating strategies, and conditioning tricks (teacher forcing, scheduled sampling) for sequential modeling while connecting to the diagnostics from earlier chapters.

Chapter 13 *Transformers and Attention* consolidates scaled dot-product attention, multi-head blocks, encoder/decoder stacks, long-context strategies (RoPE/ALiBi, FlashAttention, KV caches), and PEFT techniques.

Chapter 14 *NLP Pipelines and Responsible Deployment* links static/contextual embeddings to downstream tasks, adds bias/calibration checklists, and closes with a deployment-readiness assessment.

Chapter 15 *Soft Computing Orientation* positions fuzzy logic,

neurocomputing, probabilistic reasoning, and evolutionary search as complementary tools and introduces the running thermostat example used in Chapters 16 to 18.

Chapter 16 *Fuzzy Sets and Membership Functions* defines linguistic variables, membership design patterns, set operations, and inclusion metrics that quantify vagueness and overlap.

Chapter 17 *Fuzzy Relations and the Extension Principle* covers Cartesian products, projections, and composition operators (max–min, algebraic, Łukasiewicz) that transfer fuzzy information across universes.

Chapter 18 *Fuzzy Inference Systems* assembles complete Mamdani and Sugeno pipelines (aggregation, implication, defuzzification) and studies practical operator choices, scaling, and thermostat/autofocus examples.

Chapter 19 *Evolutionary and Population-Based Search* surveys canonical GAs, GP, CMA-ES, and Differential Evolution, emphasizing constraint handling, budget-aware population sizing, and integration with the rest of the toolkit.

Four-level taxonomy in practice

Level 1 (reactive systems): Feedback loops and associative memories that respond instantly, e.g., Hopfield dynamics in Chapter 10 or the thermostat-style fuzzy controllers introduced across Chapters 16 to 18.
Level 2 (deliberative planners): Rule-based systems that reason over an internal linguistic state before acting; see the fuzzy relation and inference machinery of Chapters 16 to 18, where conditions aggregate before a crisp recommendation is issued.
Level 3 (adaptive learners): Data-driven models that update parameters from data, spanning the ERM toolkit (Chapters 3 to 4), perceptrons/MLPs/RBFs/CNNs (Chapters 5 to 8 and Chapter 11), sequence models and Transformers (Chapters 12 to 13), SOMs (Chapter 9), and population heuristics (Chapter 19).
Level 4 (meta-cognitive agents): Algorithms that reason about their own learning loops: calibration and uncertainty estimation (Chapters 3 to 4), training diagnostics and early-stop policies (Chapter 7), alignment/PEFT tooling for Transformers (Chapter 13), and self-adaptive evolutionary strategies (Chapter 19). These illustrate early steps toward systems that refine their own policies.

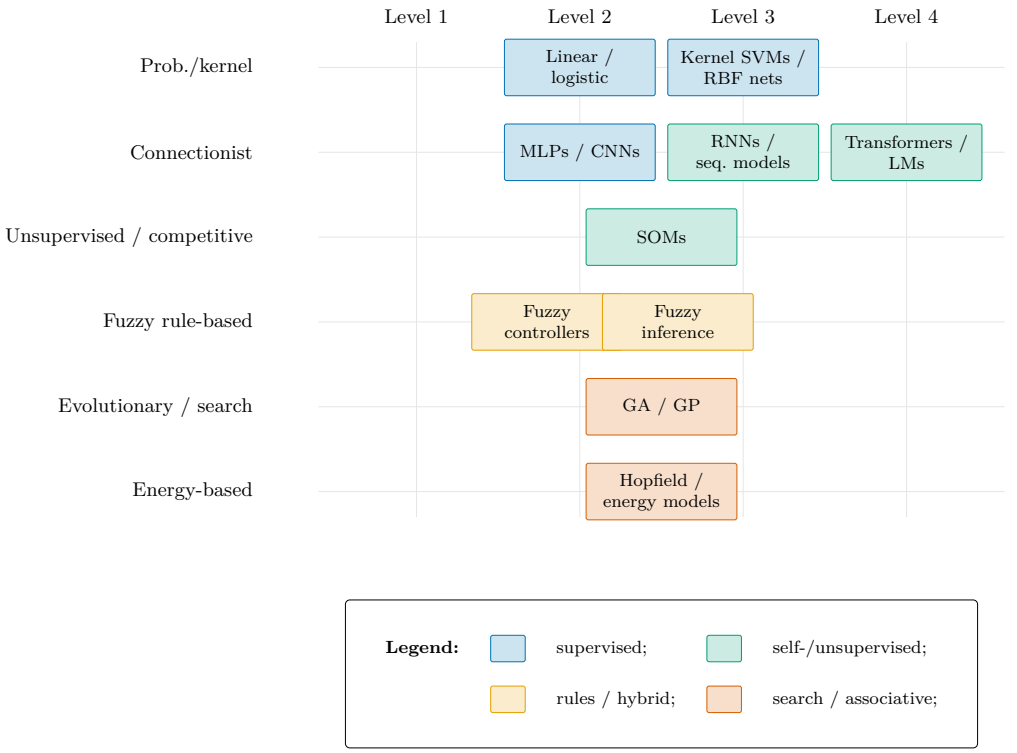

Figure 76: Schematic: Color-coded map of model families across agent level, model nature, and learning signal.

Table 11: Schematic: Big-picture view of model families across the taxonomy and learning paradigms. Each entry represents a family introduced in the book; supervision labels indicate the dominant training signal rather than strict exclusivity.

Model family	Level	Nature	Learning signal
Linear / logistic regression	2–3	probabilistic	supervised
Kernel SVMs and RBF networks	2–3	probabilistic / kernel	supervised
MLPs / CNNs	3	connectionist	supervised
RNNs / sequence models	3	connectionist	supervised / self-supervised
Transformers / attention LMs	3–4	connectionist	self-supervised
Self-organizing maps (SOMs)	2–3	unsupervised / competitive	unsupervised
Fuzzy controllers and inference systems	1–2	fuzzy rule-based	supervised / expert rules
Genetic algorithms and GP	1–3	evolutionary	search / fitness-driven
Hopfield and modern Hopfield variants	1–3	energy-based	associative / unsupervised

References

Daniel J. Amit, Hanoch Gutfreund, and Haim Sompolinsky. Storing infinite numbers of patterns in a spin-glass model of neural networks. *Physical Review Letters*, 55(14):1530–1533, 1985.

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565*, 2016.

Ronald C. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016. arXiv:1607.06450.

Wyllis Bandler and Ladislav J. Kohout. Semantics of implication operators and fuzzy relational products. *International Journal of Man-Machine Studies*, 12 (1):89–116, 1980.

Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, 2019.

Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*, volume 28, 2015.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994. doi: 10.1109/72.279181.

Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

Tolga Bolukbasi, Kai-Wei Chang, James Zou, Venkatesh Saligrama, and Adam Kalai. Man is to computer programmer as woman is to homemaker? debiasing word embeddings. In *Advances in Neural Information Processing Systems*, volume 29, pages 4349–4357, 2016.

Manuel Bronstein. *Symbolic Integration I: Transcendental Functions*. Springer, 2nd ed., 2005.

Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, 2(1):14–23, 1986.

Chi-Tsong Chen. *Linear System Theory and Design*. Oxford University Press, 3rd ed., 1999.

Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In *EMNLP*, pages 103–111, 2014.

Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

M. Cottrell and J. C. Fort. A stochastic model of retinotopy: A self organizing process. *Biological Cybernetics*, 53(6):405–411, 1986.

Michael T. Cox and A. Raja. Metareasoning: A manifesto. *AI Magazine*, 32(1): 39–54, 2011.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.

Kalyanmoy Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley, 2001.

Kalyanmoy Deb and Ram Bhushan Agrawal. Simulated binary crossover for continuous search space. In *Complex Systems*, volume 9, pages 115–148, 1995.

Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evolutionary Computation*, 6(2):182–197, 2002.

Tim Dettmers, Younes Belkada, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. In *Advances in Neural Information Processing Systems*, 2023.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, pages 4171–4186, 2019.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.

Didier Dubois and Henri Prade. *Fuzzy Sets and Systems: Theory and Applications*. Academic Press, 1980.

Didier Dubois and Henri Prade. *Possibility Theory: An Approach to Computerized Processing of Uncertainty*. Plenum Press, 1988.

Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. Wiley, 2nd ed., 2001.

Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

E. Erwin, K. Obermayer, and K. Schulten. Self-organizing maps: Ordering, convergence properties and energy functions. *Biological Cybernetics*, 67(1): 47–55, 1992.

Bernd Fritzke. A growing neural gas network learns topologies. In *Advances in Neural Information Processing Systems*, volume 7, pages 625–632, 1994.

Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Advances in Neural Information Processing Systems*, volume 29, 2016.

Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 2000. doi: 10.1162/089976600300015015.

David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

Albert Gu, Isys Johnson, and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. On calibration of modern neural networks. In *Proceedings of the 34th International Conference on Machine Learning*, 2017.

Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer, 2nd ed., 2009.

Simon Haykin. *Neural Networks and Learning Machines*. Pearson, 3rd ed., 2009.

Simon Haykin. *Adaptive Filter Theory*. Pearson, 5th ed., 2013.

Francisco Herrera and Manuel Lozano. *Fuzzy Evolutionary Computation*. Springer, 2008.

Donald R. Hill. *Studies in Medieval Islamic Technology*. Routledge, 1998.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, et al. Training compute-optimal large language models. In *arXiv preprint arXiv:2203.15556*, 2022.

John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

John J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79 (8):2554–2558, 1982.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

IEEE Global Initiative on Ethics of Autonomous and Intelligent Systems. Ethically aligned design. 1st ed. white paper, 2019. IEEE Standards Association.

Hisao Ishibuchi and Tomohiro Nakashima. *Fuzzy Multiobjective Optimization: Evolutionary and Genetic Algorithms*. Springer, 2007.

Jyh-Shing Roger Jang. ANFIS: Adaptive-network-based fuzzy inference system. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(3):665–685, 1993. doi: 10.1109/21.256541.

Kenneth A. De Jong. *Evolutionary Computation: A Unified Approach*. MIT Press, 2006.

Daniel Jurafsky and James H. Martin. Speech and language processing, 2023. Draft (3rd ed.), chapters available online.

Thomas Kailath. *Linear Systems*. Prentice Hall, 1980.

Jared Kaplan, Sam McCandlish, Tom Henighan, et al. Scaling laws for neural language models. In *arXiv preprint arXiv:2001.08361*, 2020.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *Proc. ICLR*, 2015.

Erich P. Klement, Radko Mesiar, and Endre Pap. *Triangular Norms*. Trends in Logic. Springer, 2000.

George J. Klir and Bo Yuan. *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Prentice Hall, 1995.

Teuvo Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69, 1982.

Teuvo Kohonen. *Self-Organizing Maps*. Springer, 3rd ed., 2001.

John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25:1097–1105, 2012.

Dmitry Krotov and John J. Hopfield. Dense associative memory for pattern recognition. *Advances in Neural Information Processing Systems*, 29, 2016.

Dmitry Krotov and John J. Hopfield. Large associative memory problem in neurobiology and machine learning. *Journal of Statistical Mechanics: Theory and Experiment*, page 034003, 2020.

David Krueger, Tegan Maharaj, Jörg Tiedemann, et al. Zoneout: Regularizing rnns by randomly preserving hidden activations. In *International Conference on Learning Representations*, 2017.

Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. In *Advances in Neural Information Processing Systems*, 2014.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *Proc. ICLR*, 2019.

J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

Ebrahim H. Mamdani and Sedrak Assilian. An experiment in linguistic synthesis with a fuzzy logic controller. *International Journal of Man-Machine Studies*, 7(1):1–13, 1975.

T. M. Martinetz, S. G. Berkovich, and K. J. Schulten. Neural-gas network for vector quantization and its application to time-series prediction. *IEEE Transactions on Neural Networks*, 4(4):558–569, 1993.

Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

Robert J. McEliece, Edward C. Posner, Eugene R. Rodemich, and Santosh S. Venkatesh. The capacity of the hopfield associative memory. *IEEE Transactions on Information Theory*, 33(4):461–482, 1987.

Charles A. Micchelli. Interpolation of scattered data: Distance matrices and conditionally positive definite functions. *Constructive Approximation*, 2(1): 11–22, 1986.

Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Černocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Proc. Interspeech*, 2010.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *Proceedings of ICLR*, 2013. arXiv:1301.3781.

Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1998.

Katsuhiko Ogata. *Modern Control Engineering*. Prentice Hall, 5th ed., 2010.

Jihun Park and Irene W. Sandberg. Universal approximation using radial-basis-function networks. *Neural Computation*, 3(2):246–257, 1991.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Proceedings of EMNLP*, pages 1532–1543, 2014.

John C. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. In Alexander J. Smola, Peter Bartlett, Bernhard Schölkopf, and Dale Schuurmans, editors, *Advances in Large Margin Classifiers*, pages 61–74. MIT Press, 1999.

Tomaso Poggio and Federico Girosi. Networks for approximation and learning. *Proceedings of the IEEE*, 78(9):1481–1497, 1990.

David Poole and Alan Mackworth. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, 2nd ed., 2017.

Ofir Press and Lior Wolf. Using the output embedding to improve language models. In *Conference of the European Chapter of the Association for Computational Linguistics*, pages 157–163, 2017.

Ofir Press, Noah A. Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *Proceedings of NeurIPS*, 2022.

Hubert Ramsauer, Bernhard Schäfl, Johannes Lehner, Philipp Seidl, Michael Widrich, Thomas Adler, Lukas Gruber, Markus Holzleitner, Milena Pavlovic,

Michael Sandve, Viktor Deiseroth, and Sepp Hochreiter. Hopfield networks is all you need. *International Conference on Learning Representations*, 2021.

Robert H. Risch. The problem of integration in finite terms. *Transactions of the American Mathematical Society*, 139:167–189, 1969.

Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 4th ed., 2021.

Peter J. Schreier and Louis L. Scharf. *Statistical Signal Processing of Complex-Valued Data: The Theory of Improper and Noncircular Signals*. Cambridge University Press, 2010.

Stanislau Semeniuta, Aliaksei Severyn, and Erhardt Barth. Recurrent dropout without memory loss. In *COLING*, pages 1757–1766, 2016.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations*, 2017.

Rainer Storn and Kenneth Price. Differential Evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359, 1997.

Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. In *Proceedings of ACL*, 2021. arXiv:2104.09864.

Tomohiro Takagi and Michio Sugeno. Fuzzy identification of systems and its applications to modeling and control. *IEEE Transactions on Systems, Man, and Cybernetics*, 15(1):116–132, 1985.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. In *1960 IRE WESCON Convention Record*, volume 4, pages 96–104, 1960.

Bernard Widrow and Samuel D. Stearns. *Adaptive Signal Processing*. Prentice Hall, 1985.

David Willshaw and C. von der Malsburg. How patterned neural connections can be set up by self-organization. *Proceedings of the Royal Society of London. Series B. Biological Sciences*, 194(1117):431–445, 1976.

John Yen and Reza Langari. *Fuzzy Logic: Intelligence, Control, and Information*. Prentice Hall, 1999.

Lotfi A. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, 1965.

Lotfi A. Zadeh. The concept of a linguistic variable and its application to approximate reasoning. *Information Sciences*, 8(3):199–249, 1975.

Lotfi A. Zadeh. Fuzzy logic, neural networks, and soft computing. *Communications of the ACM*, 37(3):77–84, 1994.

Lotfi A. Zadeh. What is soft computing? *Soft Computing*, 1:1–2, 1997.

Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm. *Technical Report 103*, 2002.

A Linear Systems Primer

This appendix collects basic material on signals, linear time-invariant (LTI) systems, and state-space models. It serves as a reference for the dynamical viewpoints used in later chapters (e.g., Hopfield networks, RNNs, and control-oriented fuzzy systems).

Signals and Systems

A *signal* is a mapping from an index set (typically time or space) into a set of values that encode a physical or abstract quantity. Formally, a continuous-time *scalar* signal is a function $x : \mathbb{R} \to \mathbb{R}$ (or $\mathbb{C}$), while a continuous-time *vector* signal is $x : \mathbb{R} \to \mathbb{R}^n$ (or $\mathbb{C}^n$). Discrete-time signals are defined analogously on $\mathbb{Z}$. Signals may be deterministic, stochastic, scalar, or vector-valued depending on the context.

A *system* is an operator $\mathcal{T}$ that maps an input signal space $\mathcal{X}$ to an output signal space $\mathcal{Y}$, i.e., $y = \mathcal{T}\{x\}$. Systems are characterized by properties such as linearity, time-invariance, causality, and stability; determining which of these properties hold tells us which analytical tools (Fourier analysis, state-space models, etc.) are applicable.

Linear Time-Invariant Systems

LTI systems are a central class of models. They satisfy:

- **Linearity:** For any inputs $x_1(t)$, $x_2(t)$ and scalars a_1, a_2,

$$\mathcal{S}[a_1 x_1(t) + a_2 x_2(t)] = a_1 \mathcal{S}[x_1(t)] + a_2 \mathcal{S}[x_2(t)].$$

- **Time-invariance:** If the input is shifted in time by τ, the output is shifted by the same amount:

$$\mathcal{S}[x(t-\tau)] = y(t-\tau),$$

  where $y(t) = \mathcal{S}[x(t)]$.

Impulse Response and Convolution

The behavior of an LTI system is completely characterized by its *impulse response* $h(t)$, defined as the output when the input is a Dirac delta function $\delta(t)$:

$$h(t) = \mathcal{S}[\delta(t)].$$

For any input $x(t)$, the output $y(t)$ is given by the convolution integral

$$y(t) = (x * h)(t) = \int_{-\infty}^{\infty} x(\tau)\, h(t-\tau)\, d\tau. \tag{A.1}$$

In discrete time the integral is replaced by a sum over integer indices.

Frequency-Domain Representation

The Fourier transform is a standard tool for analysing signals and LTI systems in the frequency domain. For a signal $x(t)$, the Fourier transform $X(f)$ is

$$X(f) = \int_{-\infty}^{\infty} x(t)\, e^{-j2\pi f t}\, dt.$$

Under suitable regularity conditions, convolution in time corresponds to multiplication in frequency:

$$Y(f) = H(f)X(f),$$

where $H(f)$ is the transform of $h(t)$.

State-Space Models and Transfer Functions

Many dynamical systems in this book are expressed in continuous-time state-space form:

$$\frac{d\mathbf{x}(t)}{dt} = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t), \tag{A.2}$$

$$\mathbf{y}(t) = \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t), \tag{A.3}$$

where $\mathbf{x}(t) \in \mathbb{R}^n$ is the state, $\mathbf{u}(t) \in \mathbb{R}^m$ the input, and $\mathbf{y}(t) \in \mathbb{R}^p$ the output.

Homogeneous solution. For the zero-input system, the state evolves as

$$\mathbf{x}(t) = e^{\mathbf{A}t}\mathbf{x}(0), \tag{A.4}$$

where $e^{\mathbf{A}t}$ is the matrix exponential

$$e^{\mathbf{A}t} = \sum_{k=0}^{\infty} \frac{(\mathbf{A}t)^k}{k!}. \tag{A.5}$$

Key properties include $e^{\mathbf{A}0} = \mathbf{I}$ and $\frac{d}{dt}e^{\mathbf{A}t} = \mathbf{A}e^{\mathbf{A}t} = e^{\mathbf{A}t}\mathbf{A}$. If $\mathbf{A}$ is diagonalizable, $e^{\mathbf{A}t}$ can be computed efficiently via eigen-decomposition.

Forced response. With input $\mathbf{u}(t)$, the solution is

$$\mathbf{x}(t) = e^{\mathbf{A}t}\mathbf{x}(0) + \int_0^t e^{\mathbf{A}(t-\tau)}\mathbf{B}\mathbf{u}(\tau)\, d\tau, \tag{A.6}$$

which mirrors the convolution expression for LTI systems: the kernel $e^{\mathbf{A}(t-\tau)}$ plays the role of a matrix-valued impulse response.

Transfer function. Taking the Laplace transform of (A.2) with zero initial conditions yields

$$s\mathbf{X}(s) = \mathbf{A}\mathbf{X}(s) + \mathbf{B}\mathbf{U}(s), \tag{A.7}$$

$$\mathbf{Y}(s) = \mathbf{C}\mathbf{X}(s) + \mathbf{D}\mathbf{U}(s). \tag{A.8}$$

Solving for $\mathbf{X}(s)$ gives

$$\mathbf{X}(s) = (s\mathbf{I} - \mathbf{A})^{-1}\mathbf{B}\mathbf{U}(s), \tag{A.9}$$

and substituting into the output equation produces the transfer function matrix

$$\mathbf{G}(s) = \mathbf{C}(s\mathbf{I} - \mathbf{A})^{-1}\mathbf{B} + \mathbf{D}. \tag{A.10}$$

This compactly describes the input–output behavior of the LTI system in the Laplace domain.

Further Reading

- Kailath, T. (1980). *Linear Systems*. Prentice Hall.
- Chen, C.-T. (1999). *Linear System Theory and Design*. Oxford University Press.
- Ogata, K. (2010). *Modern Control Engineering*. Prentice Hall.

B Kernel Methods and Support Vector Machines

This appendix is a concise reference for the "kernel/SVM" thread that appears throughout the book (hinge losses in Chapter 3, RBF feature maps in Chapter 8, and the geometry of margins). The goal is not to be exhaustive, but to make the notation and the relationship between explicit (finite) feature maps and implicit (kernel) feature maps unambiguous.

Kernel trick and Gram matrices

Let $\phi : \mathbb{R}^d \to \mathcal{H}$ be a (possibly high-dimensional) feature map into an inner-product space $\mathcal{H}$. A *kernel* is a symmetric, positive semidefinite function

$$k(\mathbf{x}, \mathbf{z}) \triangleq \langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle_{\mathcal{H}}.$$

Given training points $\{\mathbf{x}_i\}_{i=1}^N$, the *Gram matrix* $K \in \mathbb{R}^{N \times N}$ has entries $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$; by construction, K is symmetric and positive semidefinite.

Kernel ridge regression (KRR)

Kernel ridge regression fits a function of the form

$$f(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i \, k(\mathbf{x}_i, \mathbf{x}),$$

where the coefficient vector $\boldsymbol{\alpha} \in \mathbb{R}^N$ solves

$$(K + \lambda I)\boldsymbol{\alpha} = \mathbf{y}. \tag{B.1}$$

Here $\lambda > 0$ regularizes the solution and stabilizes the linear system when K is ill-conditioned. This is the fully kernelized analogue of ridge regression in a

finite design matrix Φ. Chapter 8's dual viewpoint shows that a primal RBF network with centers at all data points ($M = N$) recovers this same predictor under an RBF kernel.

Soft-margin SVMs (primal and kernelized form)

For binary labels $y_i \in \{-1, +1\}$, the (linear) soft-margin SVM solves (Cortes and Vapnik, 1995)

$$\min_{\mathbf{w},b,\boldsymbol{\xi}} \quad \frac{1}{2}\|\mathbf{w}\|_2^2 + C\sum_{i=1}^{N}\xi_i \tag{B.2}$$

$$\text{s.t.} \quad y_i(\mathbf{w}^\top \mathbf{x}_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0. \tag{B.3}$$

The parameter $C > 0$ trades margin size against slack violations. The decision function is $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$, with classification by $\text{sign}(f(\mathbf{x}))$. In the kernelized form, $\mathbf{w}$ is never formed explicitly; instead,

$$f(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i y_i \, k(\mathbf{x}_i, \mathbf{x}) + b, \tag{B.4}$$

where many coefficients α_i become zero (only *support vectors* remain).

How kernels relate to RBFNs (and when to use which)

- **RBFN (explicit features):** choose $M \ll N$ centers and widths, build $\Phi \in \mathbb{R}^{N\times M}$, and solve a regularized linear system in M unknowns. This is efficient when you can afford explicit features and want direct control over locality and model size.
- **Kernel method (implicit features):** work directly with $K \in \mathbb{R}^{N\times N}$, which corresponds to an implicit feature map of potentially very high dimension. This is attractive for small-to-medium N when the kernel encodes a useful inductive bias, but training and storage scale poorly with N unless approximations are used.
- **Nyström / low-rank approximations:** choose M landmark points and project into a rank-M space, recovering an explicit finite basis that connects the kernel view back to the RBFN picture in Chapter 8.

C Course Logistics

This appendix consolidates administrative information that previously appeared in scattered subsections of Chapter 1. It is intended to remain stable across offerings and can be skimmed or skipped by readers focused purely on the technical material.

Materials

This book is self-contained. It does not rely on external companion resources (such as code bundles, figure packs, or solution sets). When the book is used in a course offering, offering-specific documents (syllabus, deadlines, problem sets) are distributed through the local course channel and may change from term to term.

Communication

Questions and feedback can be handled via email or a forum if one is announced. Office hours, if any, will be communicated alongside the course materials.

Assessment Overview

Assignments (individual or groups up to three), examinations (if applicable), and self-check exercises interleaved with chapters. Exact dates and policies depend on the offering and will be communicated with the course materials.

Policies

Submission windows, late policies, and academic integrity guidelines are offering-specific and should be consulted in the local course documentation.

C.1 Using this book in ECE 657

When this book is used in ECE 657 at the University of Waterloo, the following apply:

- **Communication/support:** Course announcements (including clarifications and errata relevant to an offering) are distributed through the course channel; an official forum or mailing list may be announced for Q&A. Office hours (if any) are posted with the term schedule.

- **Assessment structure:** Problem sets (solo or teams of up to three), quizzes, and two term exams are typical. Weighting and deadlines are confirmed in the term syllabus; use the most recent syllabus if numbers differ from prior offerings.
- **Pacing:** Condensed offerings do not include a formal reading week; exam weeks typically suppress new material. Follow the posted weekly schedule in the syllabus.
- **Accessibility:** Students requiring accommodations should contact AccessAbility Services and notify the instructor early so quiz/exam windows and deadlines can be adjusted. Captions and alternative formats are provided on request when videos accompany the course materials.
- **Integrity:** Follow the University of Waterloo Academic Integrity policy. Credit collaborators appropriately and avoid sharing solutions outside approved groups.

D Notation collision index

This appendix lists the most common *notation collisions* in this book: symbols that appear in multiple domains (probability, optimization, deep learning) with different meanings. The goal is not to eliminate reuse—that is unrealistic—but to make the disambiguation rule explicit so reading remains fast and consistent.

Disambiguation rule (used throughout the book)

- **Function argument wins:** $\sigma(x)$ is the sigmoid; $f(\cdot)$ is an activation; $p(\cdot)$ is a density/pmf.
- **Plain scalars default to the local domain:** σ without an argument is a width/scale unless the paragraph is explicitly about the sigmoid.
- **Typography is a hint, not a guarantee:** bold symbols are vectors/matrices; subscripts usually indicate time, layer, or an index set.

Symbol	Common meanings	Where to look / how to disambiguate
σ	Sigmoid nonlinearity; standard deviation / scale; RBF width	$\sigma(x)$ is always sigmoid; plain σ is a width/scale (e.g., RBFNs) unless the section is explicitly about activations.
λ	Regularization strength; eigenvalue; Lagrange multiplier	Regularization uses λ alongside an objective; spectral topics use λ with matrices/operators; constraints use λ as a multiplier.
p	Probability / density; padding (CNNs); momentum/parameter name in code	$p(y \mid x)$ is probability; convolution padding is stated as p in the output-size formula with n, k, s.
L	Number of layers; sequence length; loss	Layer count uses L with indices $l = 1, \ldots, L$; loss is $\mathcal{L}$; sequence length is usually T.
t	Time index; target vector/component	Time is t with sequences x_t, h_t; targets use $\mathbf{t}$ (bold) or t_k in loss definitions.
h	Hypothesis function h_θ; hidden state h_t / hidden units h	Hypotheses appear as $h_\theta(\cdot)$ in supervised chapters; hidden state uses a time index h_t; hidden width uses a dimension symbol (e.g., h or d_h) in network-shape context.

Editorial note. If you find a collision that slows reading, add a short local reminder at the first use in that chapter and (if it repeats across chapters) extend this index rather than creating a one-off rule.