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

I have taught courses that engage artificial intelligence both directly—through machine learning and neural networks—and indirectly, at the level of data collection and system design. Over time, I noticed a recurring gap. While the literature offers many excellent treatments of individual techniques, it rarely provides a coherent path that guides students from foundational notions of intelligence to the 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 attempts to make that perspective explicit and durable.

The manuscript evolved into a concise graduate companion that blends the original ECE 657 course voice 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 searching 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 im-

portant piece in a larger mosaic rather than the organizing principle. I iterated on the syllabus—moving topics, adding or removing chapters, and tightening mathematical through-lines—toward a narrative that is coherent, broad in coverage, and implementable in engineering practice.

At the time of this first edition (2026), 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 need not consult that appendix.

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 near the front of the book in *Notation and Conventions*.

The goal 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)
$y_{\text{reg}} \in \mathbb{R}$	Regression target (continuous)
$y_{\text{bin}} \in \{0, 1\}$	Binary class label (Bernoulli outcome)
$\hat{y}$	Model prediction
$\mathcal{D}$	Dataset or feasible domain
$\mathcal{L}$	Loss function (objective)
$\sigma(z)$	Sigmoid function $1/(1 + e^{-z})$
$\tanh(z)$	Hyperbolic tangent activation
$\text{ReLU}(z)$	$\max(0, z)$ 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)
$\text{softmax}(z)$	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
$\text{Var}[\cdot]$	Variance
$\text{diag}(\cdot)$	Diagonal matrix formed from a vector
$\odot$	Hadamard (elementwise) product
$\phi(\cdot)$	Feature map; in kernels, $k(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^\top \phi(\mathbf{z})$
$\mathbf{1}$, $\mathbf{I}$	$\mathbf{1}$: all-ones vector; $\mathbf{I}$: 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; when a symbol is reused, 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 symbols (e.g., 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 using a plain superscript 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 (matching the row-major dataset convention used throughout the book): a one-hot row vector $\mathbf{x} \in \{0, 1\}^{1\times|V|}$ selects a row via $\mathbf{xE} \in \mathbb{R}^{1\times d}$.
- 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

Reading aids: how to read the visuals and what to ask

- **How to read the visuals:** captions state what each color/line style represents (training vs. validation, safe vs. heuristic transformations, etc.), and table captions mention the chapter(s) they support so you can jump back quickly when a later chapter references them.
- **Four recurring questions:**
  1. What is the core scientific idea, and how does it relate to earlier material?
  2. Which methodological cautions should a practitioner keep close at hand?
  3. How do the accompanying figures or derivations anchor those ideas visually?
  4. 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

Intelligent systems are engineered artifacts that perceive, reason, and act under constraints. This chapter sets the shared vocabulary (historical context, core definitions, recurring design themes), and Figure 1 shows how the strands connect and where to enter.

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.

Use this chapter as a standing reference: later tools reuse the same system vocabulary and the same habit of checking assumptions against constraints. The point is continuity, not memorization: each later chapter instantiates the same design loop with different model families.

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.

What we mean by a model. In this book, a *model* is any system that maps observed information to a prediction about an outcome. Sometimes the prediction is a single value (regression), and often it is a probability distribution over

possible outcomes (classification). The details change from chapter to chapter, but the contract stays the same: choose a representation, define what it means to be wrong, and then tune the parameters so the predictions improve under an honest evaluation.

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]

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

[1] Compare with the IEEE Global Initiative on Ethics of Autonomous and Intelligent Systems, *Ethically Aligned Design*, 1st ed., 2019.

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.

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 starts with sensors (for example, cameras) whose signals are converted into numeric arrays and state estimates. Reasoning uses those estimates to classify objects (stop signs, pedestrians) and predict near-future motion. Action closes the loop by issuing steering, acceleration, and braking commands that satisfy safety constraints.

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 (complementary metal-oxide-semiconductor) sensor, on-board DSP (digital signal processor)/accelerator, motorized pan/tilt for re-targeting.

Software (reasoning)

A YOLO-style object detector (e.g., YOLOv8) fine-tuned on site-specific data, fused with Kalman filters for track smoothing and MPC (model predictive control) 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 perception–reasoning–action loop, augmented by learning and adaptation. A fuller capability checklist appears later in the Key Characteristics paragraph after the system vignettes.

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 $\mathbf{x}_i(t+1) = f\big(\mathbf{x}_i(t), \{\mathbf{x}_j(t)\}_{j\in\mathcal{N}_i}\big)$, 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 patterns 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"

When I say *strong* machine intelligence, I mean something specific: the system can turn the lens inward. It does not just predict; it also keeps enough self-modeling machinery (confidence monitors, policy checks, explanation traces) to critique and revise *its own* reasoning. This is the machine becoming the subject of its own thought.
The rest of the book uses a simpler four-layer taxonomy (reactive → deliberative → adaptive → meta-cognitive). Keep this "subject of its own thought" lens in the background: it is why Level 4 systems demand extra 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 require auditable decision traces, routine self-inspection/error analysis, and bounded backtracking/self-correction inside explicit, designer-defined interfaces.

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.
- **While reading:** follow cross-referenced figures/equations and pause at the short checkpoints and worked examples.

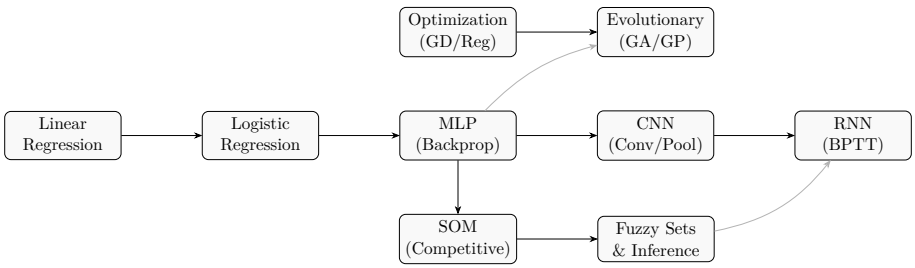

Figure 1: Roadmap of the book strands (core supervised path; SOM/fuzzy; optimization/evolutionary). It also serves as a quick prerequisite map when you want to jump ahead and later backfill foundations.

- **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

Minimum viable mastery

- 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.

Common pitfalls

- Treating the roadmap as a single linear syllabus rather than a dependency graph of prerequisites.
- Confusing "probability" with "decision": many later failures come from thresholds and costs, not from modeling.
- Letting notation drift: keep Appendix D close when symbols are reused in different chapters.

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.
- Choose a reading path using Figure 1, and write down which two chapters you will skim first (and why).

If you are skipping ahead. Keep the operational vocabulary (representation, actions, objective/goal test, and audit/checks) and the roadmap (Figure 1) in mind; later chapters assume these as the organizing lens for both symbolic and data-driven tools.

Where we head next. Chapter 2 grounds this vocabulary in a compact case study: symbolic integration as transformation search. The example shows decomposition, safe rewrites, heuristic branching, backtracking, and residual checks in one place; the supervised chapters then reinterpret the same loop with losses, parameter updates, and validation audits. Keep this continuity in view: the implementation changes, but the engineering questions (state, action, objective, verification) stay the same through fuzzy reasoning and evolutionary optimization.

2 Symbolic Integration and Problem-Solving Strategies

Chapter 1 treated "intelligence" operationally: represent a problem, choose actions, and verify or correct under constraints. Here we make that concrete with a deliberately small example—symbolic integration—where the system's actions are algebraic transformations. Figure 1 places this symbolic strand alongside the data-driven path, and we use it as a rehearsal for the same audit discipline that appears later in ERM, fuzzy inference, and evolutionary search.

We study symbolic problem solving through *integration-by-transformation*: preserve meaning while rewriting an expression into a form that exposes a solution. The goal is not a catalog of tricks, but the system pattern: explicit representations, meaning-preserving actions, heuristic branching with backtracking, and a goal test that certifies correctness.

We 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 exhaust safe steps first, then branch judiciously through the heuristic catalog while keeping enough state to backtrack.

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.

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, $F'(x) - f(x) = 0$).

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.

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.

2.2 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).

Limitations of safe transformations Safe transformations preserve meaning and allow back-substitution, but they are not a complete solution strategy: after exhausting the invertible rewrites you know, you can still end up with an integrand that does not match any template you can finish. At that point the system has to make an explicit decision: stop (declare "unknown" under the current rule set), or branch into heuristic moves that may succeed but are not guaranteed.

2.3 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.

2.4 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 nodes in the expression's abstract syntax tree (AST), 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).

2.5 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 applies the cost heuristic defined above (tree depth, nonlinearity, symbol count) to the two safe paths listed in Step 2; the substitution route typically scores lower and is tried first, with the reduction formula held as a fallback if substitution stalls.

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 $\int \sec^4 y\, dy = \tan y + \frac{1}{3}\tan^3 y + C$, confirming the first branch by an independent reduction route rather than 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.

Example: solving an integral via transformation trees The worked integral above already showed branch behavior (e.g., $x = \sin y$ vs. $x = \tanh u$) under domain constraints. We keep that example as a template and now make the search logic explicit: domain-incompatible heuristic branches are pruned, safe branches can converge to the same antiderivative, and residual checks provide the goal test at each accepted leaf.

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

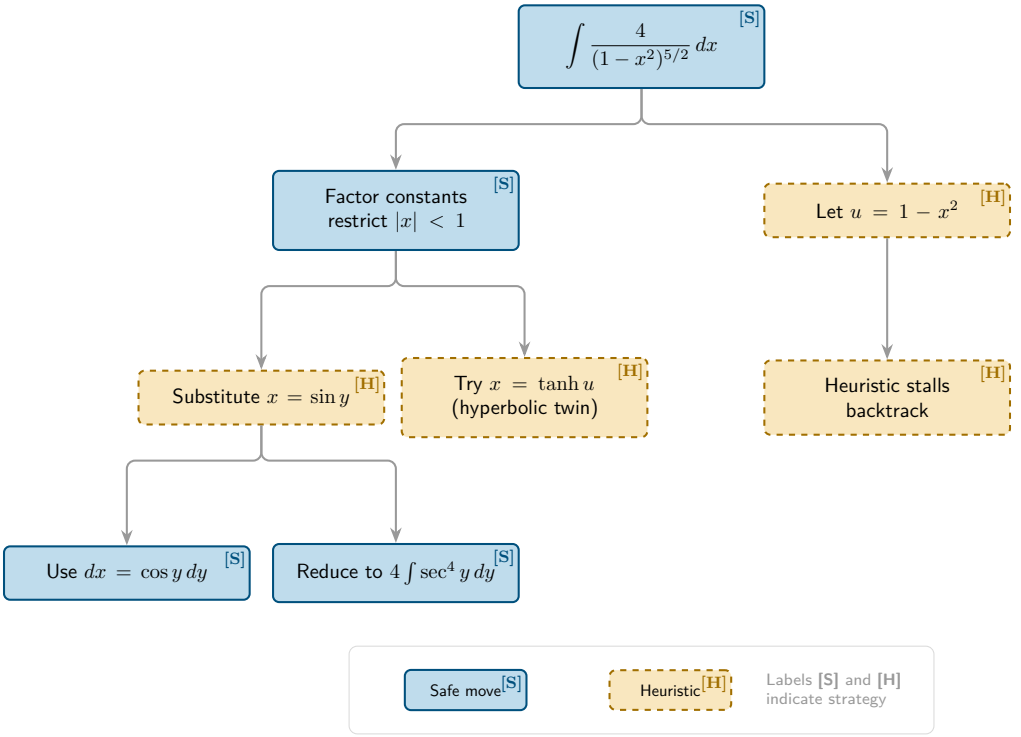

Figure 2: Transformation tree for the running example integral. Badges **[S]**/**[H]** mark safe vs. heuristic moves; the dashed branch mirrors the sine substitution. The tree view makes it easy to see where an attempt branched and where backtracking entered.

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

ulations, such as

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

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.

2.7 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.

2.8 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

Minimum viable mastery

- 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.

Common pitfalls

- Losing the system-level point in calculus detail: always name the state, action, heuristic, and goal test.
- Ignoring domains/branches: substitutions can shrink domains, introduce branch cuts, and invalidate a "correct" algebraic rewrite.
- Treating heuristics as proofs: heuristic branches must be verified (or backtracked), not trusted.

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.

If you are skipping ahead. Keep the pattern vocabulary: safe vs. heuristic moves, explicit budgets, and residual/verification checks. The data-driven chapters reuse the same discipline (objective, constraints, and validation) even though the "actions" become parameter updates; the fuzzy and evolutionary chapters reuse it again when tuning operators and rule systems under constraints.

Where we head next. For the data-driven thread (datasets, objectives, diagnostics, classification), continue to Chapters 3 and 4. For nonlinear function classes and nonconvex training dynamics, continue through Chapters 5 to 6. Later, when you reach fuzzy inference and evolutionary search, reuse this chapter's core habit: define admissible moves, track budgets, and verify outcomes against explicit criteria.

3 Supervised Learning Foundations

Chapter 2 illustrated a non-statistical lens: solve problems by transformation search, with explicit goal tests. Here we shift to the data-driven lens. Figure 1 marks this as the core supervised strand, but the continuity is intentional: we still specify state, define actions, set a budget, and verify outcomes.

When a mapping is known, we use the formula: Celsius and Fahrenheit are linked by a simple rule, and many physical laws give direct input–output relationships. The problems that motivate machine learning do not. 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). This chapter focuses on the statistical, data-driven strand: supervised learning.

In this lens, supervised learning is 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 how to fit models, compare alternatives, and check whether apparent success is real or just memorization.

Supervised learning begins with three commitments: choose 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 adjusts model parameters so predictions align with observed outputs.

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 first make the pieces explicit (data, models, and losses), then show what training looks like when it succeeds and when it fails (underfitting vs. overfitting). Next we formalize empirical risk minimization (ERM) and the main "anti-memorization" tools (regularization and validation). Finally, we work through linear regression as the first fully transparent case study where the entire pipeline is visible end to end.

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.

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.

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 independently and identically distributed (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).

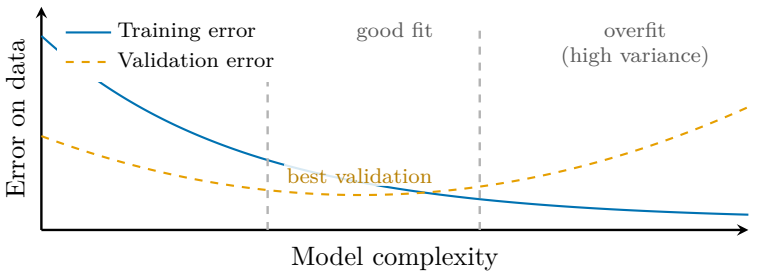

Figure 3: Schematic underfitting and overfitting as a function of model complexity. Training error typically decreases with complexity, while validation error often has a U-shape; regularization and model selection aim to operate near the minimum of the validation curve.

Figure 3 is the baseline bias-variance diagnostic used throughout the chapter.

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.

Figure 4 summarizes the geometric reason L1 and L2 penalties behave differently.

Figure 5 provides the coefficient-path view used when tuning sparsity strength.

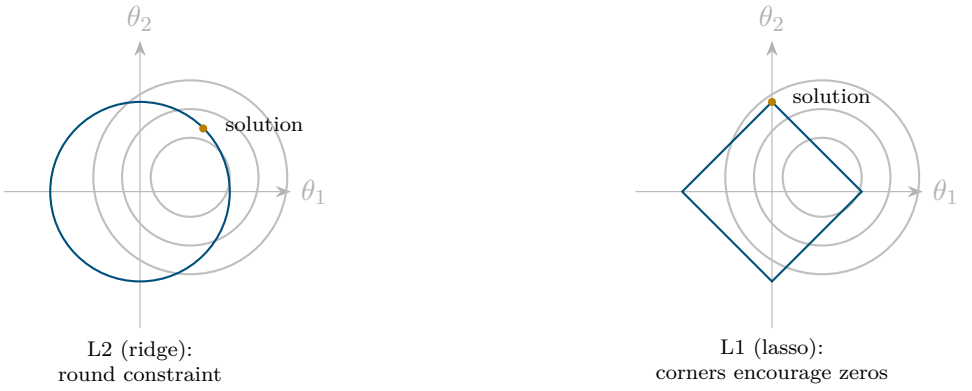

Figure 4: Why L1 promotes sparsity (geometry). Minimizing loss subject to an L2 constraint tends to hit a smooth boundary; an L1 constraint has corners aligned with coordinate axes, so tangency often occurs where some coordinates are exactly zero.

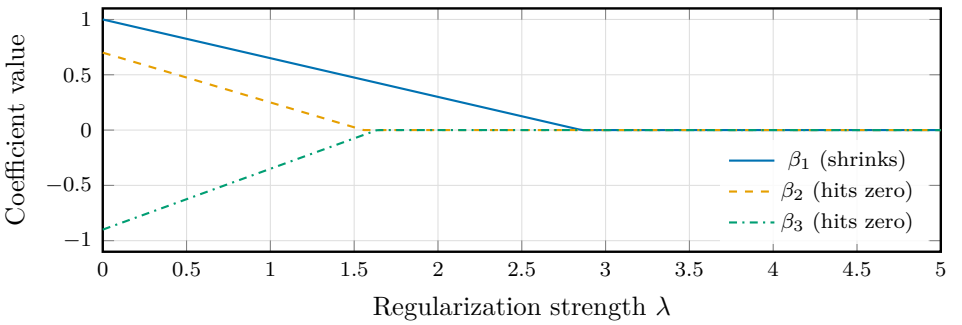

Figure 5: Illustrative lasso path as the regularization strength increases. Coefficients shrink, and some become exactly zero, yielding sparse models; the shrinkage order depends on feature correlation and scaling.

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:

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

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.

Figure 7 compares common regression losses used in this section.

Table 2: Common losses and typical use (reference for Chapters 3 to 5). Match the loss to your modeling assumptions (noise, margins) and to the downstream decision metric you care about.

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).

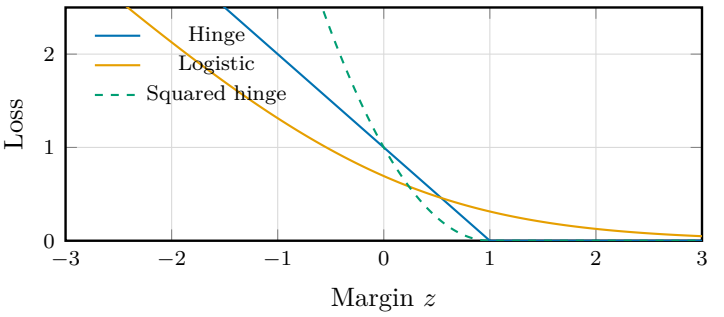

Figure 6: Classification losses as functions of the signed margin z. The curves highlight how different losses treat confident mistakes and near-boundary points.

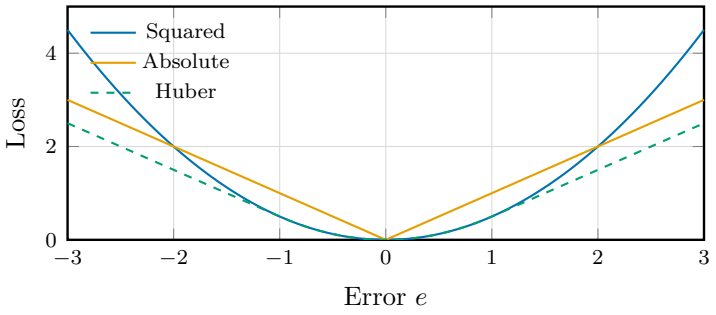

Figure 7: Regression losses versus prediction error. The Huber loss transitions from quadratic to linear, reducing sensitivity to outliers while staying differentiable near zero.

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.

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).
- **Reporting discipline:** log data split policy, seeds, and selection criteria; Appendix E defines the book-wide template.

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.

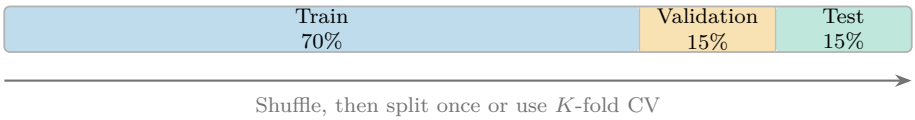

Figure 8: Dataset partitioning into training, validation, and test segments. Any resampling scheme should preserve disjoint evaluation data; when classes are imbalanced, shuffle within strata so each split reflects the overall class mix.

Figure 8 lays out the split protocol for the train/validation/test workflow.

A concrete toy task

As a reference point, keep in mind 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).

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

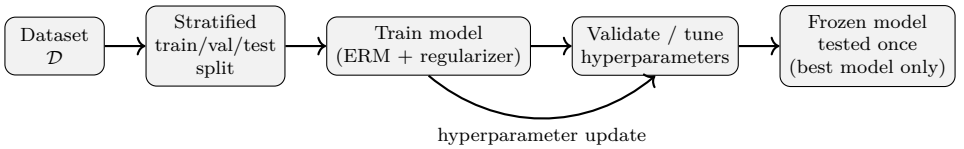

Figure 9: Mini ERM pipeline (split once, iterate train/validate, then test only the best model on the held-out set). This keeps tuning decisions separate from final reporting.

Figure 11 anchors the calibration and capacity diagnostics in this aside.

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

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.

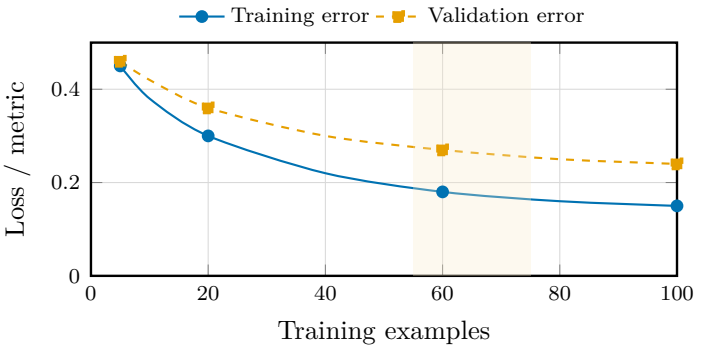

Figure 10: Illustrative learning curves reveal under/overfitting: the validation curve flattens while additional data continue to decrease training error only marginally. A shaded patience window marks when early stopping would halt if no validation improvement occurs.

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.

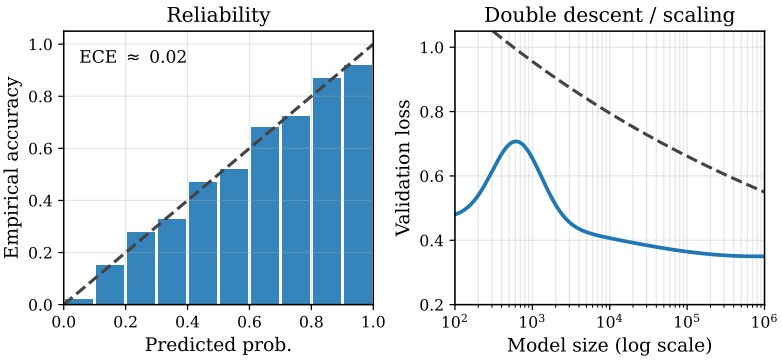

Figure 11: Calibration and capacity diagnostics. Left: reliability diagram with binned predicted probabilities vs. empirical accuracy; Expected Calibration Error (ECE) measures deviation from the diagonal. Right: illustrative double-descent risk vs. model size (log-scale on the x-axis); the dashed line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

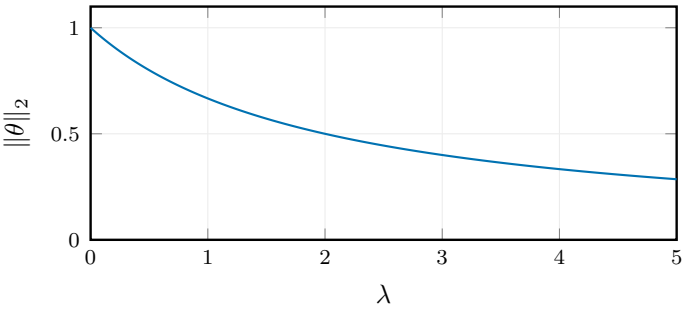

Figure 12: Ridge regularization shrinks parameter norms as the penalty strength increases. This controls variance without forcing sparsity.

gression keeps the same pipeline but makes each component explicit enough to inspect end to end.

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 estimating $\boldsymbol{\beta}$ from $(\mathbf{X}, \mathbf{y})$ so predictions align with observed targets under the chosen objective. Because least squares is convex with a closed form, repeating the fit on the same data returns the same solution (up to numerical tolerance). That determinism is the teaching benefit: it lets us isolate the roles of loss design, optimization, regularization, and validation before moving to models where those roles are harder to disentangle.

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

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

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 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:

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

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:

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

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".

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:

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

The gradient is simple,

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

so gradient descent is explicit: $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \eta \mathbf{X}^\top (\mathbf{X}\boldsymbol{\beta} - \mathbf{y})$. This same loop reappears

later when the model is no longer linear and the loss is no longer quadratic.

Least squares is convex and satisfies the normal equations:

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

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

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

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

Minimum viable mastery

- 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.

Common pitfalls

- Tuning on the test set (or peeking repeatedly) turns the test set into training data.
- Leakage through preprocessing: fit scalers/encoders/imputers on the training split only, then apply to val/test.
- Over-interpreting a single metric: use learning curves and slice audits, not only one headline number.

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.

If you are skipping ahead. Keep the ERM loop and split hygiene close: model class, loss/regularizer, optimizer, and a clean train/val/test protocol. Those ideas are assumed immediately in Chapter 4 and reused throughout the neural chapters.

Where we head next. Chapter 4 extends this ERM setup from continuous targets to discrete labels: outputs become class probabilities, the loss becomes cross-entropy, and evaluation adds confusion matrices, ROC/PR curves, and threshold selection. Keep this chapter's audit loop close; the same split hygiene and validation logic are reused in later neural chapters and remain relevant when tuning fuzzy and evolutionary systems.

4 Classification and Logistic Regression

Chapter 2 illustrated intelligence as transformation search with explicit goal tests. 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, and Figure 1 places it on the core supervised path.

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.

Design motif

Same loop, different likelihood. Logistic regression keeps the linear score, but models a Bernoulli outcome; the loss becomes a negative log-likelihood (binary cross-entropy).

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 is unchanged, but the likelihood/loss pair changes to Bernoulli plus 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.

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.

4.2 Classification problem statement

With that setup in place, we now formalize the classification target. Unlike regression, where y is continuous, classification predicts a *discrete label*. We 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: the class-conditional densities and priors are usually unknown, and estimating them reliably (especially in high dimensions) is itself a modeling problem. Practical classifiers therefore approximate Bayes' rule by making simplifying assumptions (a generative model) or by learning the posterior $P(y \mid \mathbf{x})$ directly (a discriminative model).

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 linear discriminant analysis (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.

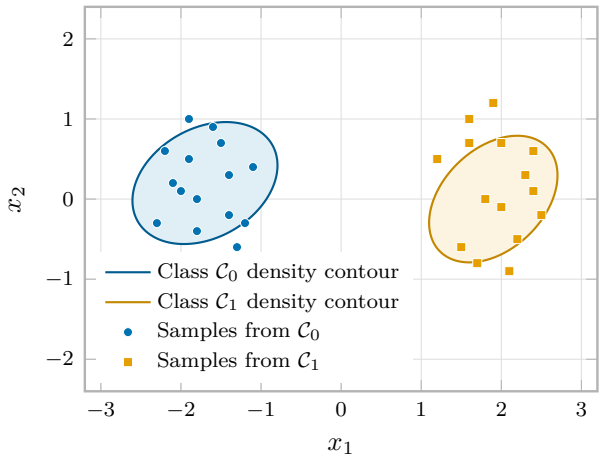

Figure 13: Synthetic binary dataset built from two anisotropic Gaussian clusters; shaded ellipses hint at the underlying density while the scattered samples are reused throughout the running examples. It helps build intuition for how decision boundaries relate to class geometry before fitting discriminative models.

As shown in Figure 15, linear scores (logits), the binary cross-entropy (BCE) loss shape, and regularization strength appear in one compact diagnostic view.

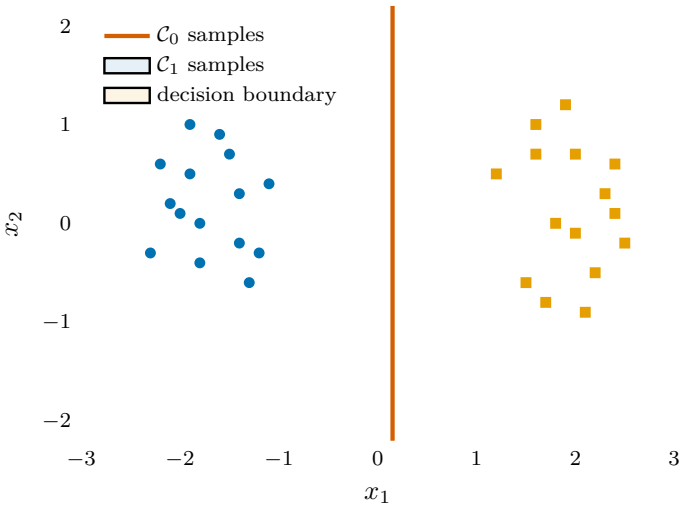

Figure 14: Bayes-optimal boundary for two Gaussian classes. Equal covariances and similar priors (LDA setting) yield a linear separator; unequal covariances yield a quadratic boundary. The boundary is near the equal-posterior line (vertical, pink); left/right regions map to predicted classes R0 and R1.

Figure 18 provides the low-data MAP-versus-MLE comparison used in this discussion.

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,

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

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 Binary labels and $\pi(\mathbf{x})$ were defined in Section 4.2; here we focus on modeling the log-odds with a linear predictor.

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"?

Logistic is the link function: a sigmoid takes a real score and turns it into a probability in $[0, 1]$. And *regression* is what remains linear: the *log-odds* (the logit) is modeled as a linear function of the features.
The model outputs a probability. The class label comes later, as a decision rule: threshold the probability in the binary case (or compare 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

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

so the log-likelihood is

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

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

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

The Hessian has the form $\mathbf{X}^\top \mathbf{W} \mathbf{X}$ where $\mathbf{W} = \operatorname{diag}(p_i(1 - p_i)) \succeq 0$, 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/iteratively reweighted least squares (IRLS) in moderate dimensions). Figure 16 is a convex quadratic toy that re-

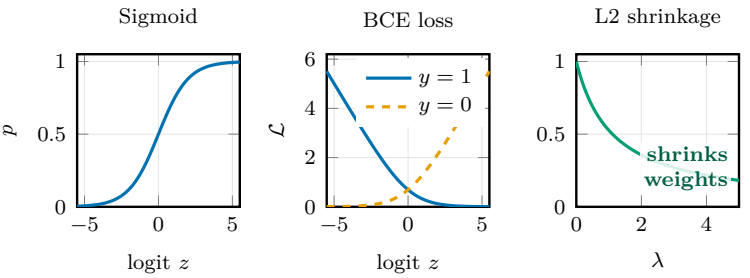

Figure 15: The sigmoid maps logits to probabilities (left). The binary cross-entropy (negative log-likelihood) penalizes confident wrong predictions sharply (middle). Regularization typically shrinks parameter norms as the penalty strength increases (right).

minds us what an optimization trajectory looks like when we minimize a smooth objective: step size and conditioning shape how quickly iterates contract toward the minimizer.

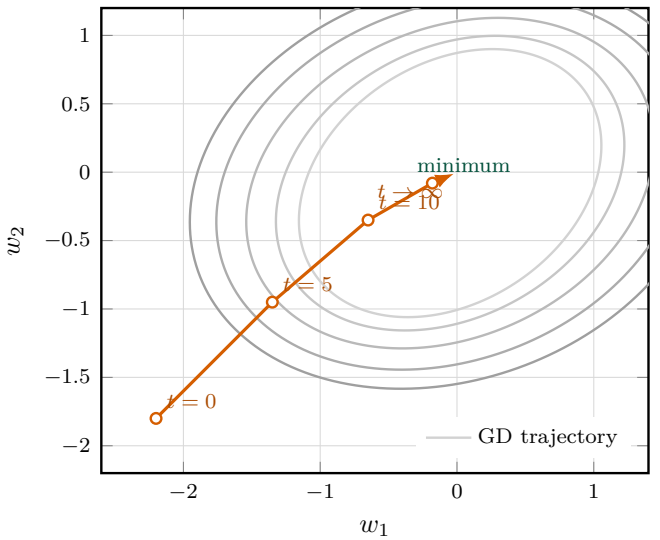

Figure 16: Gradient-descent iterates contracting toward the minimizer of a convex quadratic cost (schematic). Ellipses are level sets; arrows show the "steepest descent along contours" direction. Poor conditioning stretches the ellipses and slows convergence.

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.

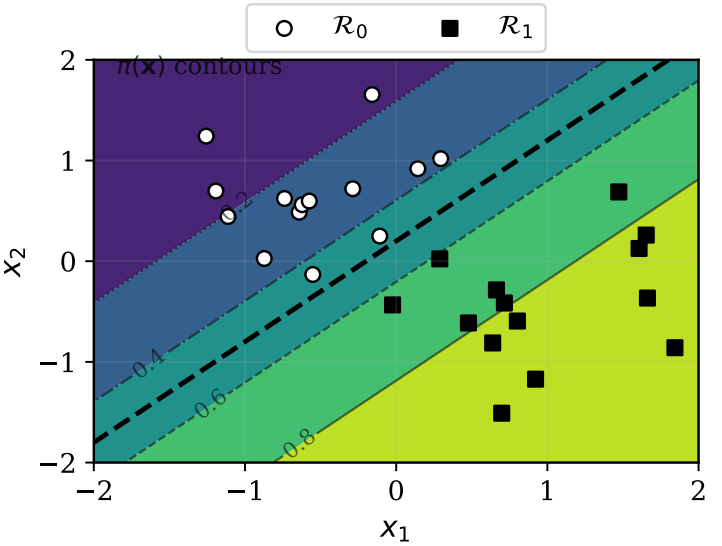

Figure 17: Illustrative logistic-regression boundary. The dashed line marks the linear decision boundary at probability 0.5; labeled contours show how the posterior varies smoothly with margin, enabling calibrated decisions and adjustable thresholds.

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.

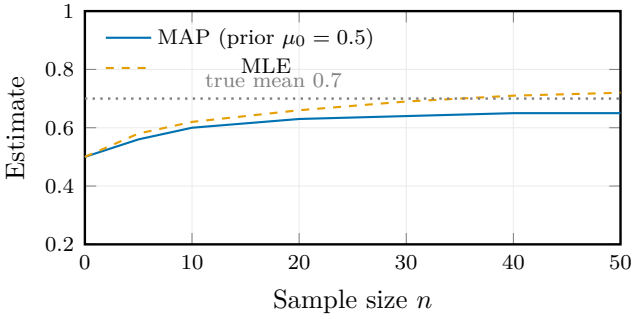

Figure 18: MAP estimates interpolate between the prior mean and the data-driven MLE. As the sample size grows, the MAP curve approaches the true mean; with little data, the prior dominates.

4.6 Confusion Matrices and Derived Metrics

Abbreviation note. ROC = receiver operating characteristic; PR = precision–recall; AUC = area under the curve; ECE = expected calibration error. We use AUROC/AUPRC for the corresponding AUCs.

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.

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.

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.

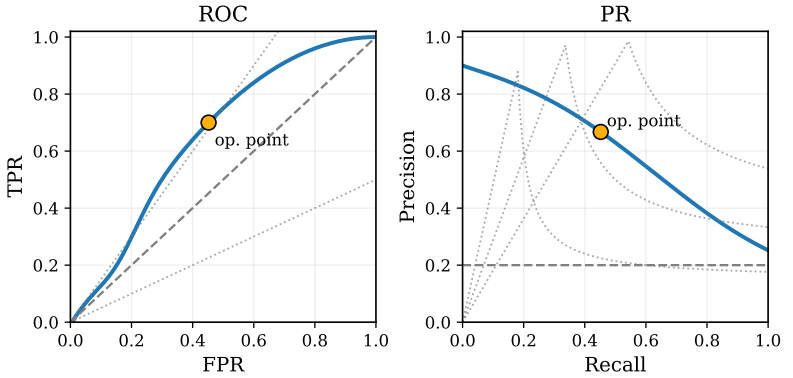

Figure 19: ROC and PR curves with an explicit operating point. Left: ROC curve with iso-cost lines; right: PR curve with a class-prevalence baseline and iso-F1 contours. Together they visualize threshold trade-offs and calibration quality.

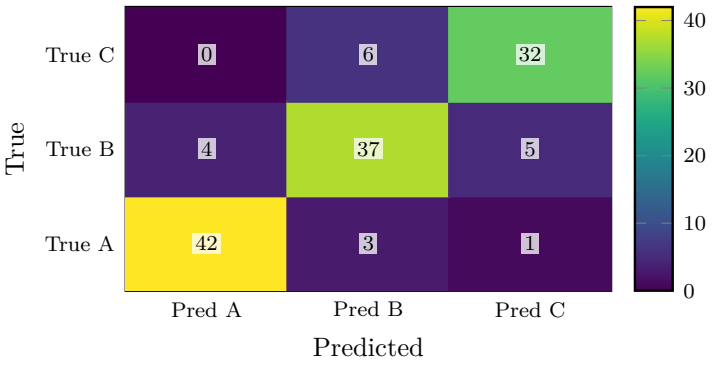

Figure 20: Confusion matrix for a three-class classifier; diagonals dominate, indicating strong accuracy with modest confusion between classes B and C.

Key takeaways

Minimum viable mastery

- 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.

Common pitfalls

- Confusing scores with decisions: choose thresholds using costs and PR/ROC trade-offs, not a default 0.5.
- Reporting only AUC: check calibration when probabilities are used downstream (reliability/ECE).
- Letting imbalance hide failure: stratify splits and audit per-class performance, not only overall accuracy.

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: Handling class imbalance for logistic models (Chapter 4 reference table). A compact menu of resampling, weighting, and thresholding strategies for rare positives.

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.

If you are skipping ahead. Keep three ideas: (i) Bayes-optimal boundaries are a conceptual anchor, (ii) logistic regression is the first practical probabilistic baseline, and (iii) evaluation depends on costs, thresholds, and calibration. Those recur throughout the neural and deployment chapters and remain important whenever later chapters output scores that must be turned into decisions.

Where we head next. Logistic regression is still linear at the boundary. Chapter 5 introduces trainable neuron models, and stacking nonlinear units then removes that linear ceiling and sets up multilayer networks plus backpropagation. Carry this chapter's decision discipline forward: probability quality, threshold choice, and class-imbalance handling remain core checks even when the model family changes.

Part I takeaways

- Intelligence as engineered self-correction: represent state, choose actions, verify outcomes.
- Two recurring toolkits: safe vs. heuristic moves (search) and ERM (model–loss–optimize–audit).
- Classification as a probabilistic decision problem: Bayes optimality and calibrated scores precede thresholds.
- Reading paths are a dependency graph: later chapters reuse diagnostics, notations, and audit habits introduced here.

Part II: Neural networks, sequence modeling, and NLP

How to read Part II (rolling window)

- **One loop, many architectures:** define a loss, run a forward pass, cache intermediates, backprop gradients, validate, then iterate.
- **Two bottlenecks reappear:** expressivity (can the model represent the boundary?) and trainability (can gradients reach the parameters?).
- **Continuity checks:** watch shapes and caching in feedforward models, watch masking in sequence models, and keep calibration/slice audits alongside accuracy.

Part II bridge: from ERM to trainable architectures

Part I treated learning as disciplined self-correction: represent the problem, define what "wrong" means (loss), improve systematically (optimization), and verify with honest splits and diagnostics. In Part II we keep that loop, but we make a second design choice explicit: we choose architectures that make learning possible at scale.

The guiding question becomes: "given this computation graph, can signals flow through it during both the forward pass (information) and the backward pass (gradients)?" That is why we begin with the perceptron, build to multi-layer networks and backpropagation, then add structure that matches common data: convolution for images, recurrence and associative memory for state and recall, and attention for content-based retrieval in language and other sequences.

Read the repeated ideas as purposeful: the ERM loop stays fixed, while the model class changes the kinds of generalization you can get and the kinds of bugs you can accidentally introduce (shape and caching mistakes, masking mistakes, and evaluation leakage).

Payoffs to watch for: (i) backprop as reusable local error signals, (ii) inductive bias as the reason CNNs, RNNs, and attention behave differently under the same ERM procedure, and (iii) auditing beyond accuracy as the difference between a model with a low loss and a system that makes good decisions.

5 Introduction to Neural Networks

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

Learning Outcomes

After this chapter, you should be able to:

- Describe the core ingredients of neural networks (architecture, activations, learning).
- Explain at a high level why multilayer perceptrons rely on smooth activations and gradient-based training (Chapters 6 and 7).
- Identify common pitfalls (saturation, poor initialization) and basic remedies.

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

Neural networks borrow a simple but powerful idea from biology: complex behavior can emerge from many simple units interacting through many simple connections. The goal is not to model the brain in detail, but to steal an engineering abstraction we can write down and implement: signals flow through a network of units, and learning adjusts connection strengths so the overall system changes its behavior.

Neurons and Neural Activity A biological neuron can be viewed as a processing unit that receives multiple inputs, integrates them, and produces an output if certain conditions are met. In the simplified picture we use here, the key parts are:

- **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.

Incoming signals can excite or inhibit the neuron. When the combined influence crosses a threshold, the neuron "fires" and sends an impulse down the axon to connected neurons. Real neurons are richer than this sketch (timing, frequency, and graded effects matter), but the abstraction is enough to motivate an artificial unit that combines inputs, applies a nonlinearity, and emits an output.

Complexities and unknowns (and what we steal anyway). Real neural tissue is a biophysical system: spikes, timing, chemistry, adaptation, and many interacting feedback loops. We do not pretend to model that faithfully here. Instead, we borrow an engineering abstraction that has two virtues: (i) it is composable, and (ii) it gives us a learning rule we can write down, debug, and scale.

So what do we *keep* from biology? The idea that many small units, each doing a simple computation, can be wired into a larger system whose behavior is richer than any single unit. And what do we *let go* of? The detailed mechanisms (spike timing, ion channels, and biological realism) that matter for neuroscience but are not necessary to build useful learning machines.

A good analogy is circuit design: you can build powerful systems out of simple, idealized components without simulating electron physics at every wire. In the same spirit, we keep the compositional story (many units, many connections) and insist that the update rule is explicit and learnable. This is a recurring theme in the book: we look to nature for candidate mechanisms behind intelligent behavior, describe them in scientific terms, and then simplify them until they become concrete engineering problems we can test and improve.

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.

Where this fits in Part II. This chapter starts the neural strand by introducing the perceptron neuron (defined below) as the basic trainable unit: a weighted sum followed by an activation/threshold, together with the first learning rules (mistake-driven and gradient-driven). The point is not to model biology literally; the point is to steal an engineering abstraction we can write down, debug, and scale.

Part II local roadmap: what stays the same, what changes

What stays the same. We still follow the empirical risk minimization (ERM) loop: choose a representation, define a loss, optimize, then verify with honest evaluation and diagnostics. As we proceed, you will find that this theme generalizes across many of the algorithms we discuss.

What changes (the design space). Instead of choosing one linear model, we build systems by composing many simple units. Used on its own, a perceptron-like unit still induces a linear decision boundary in the original feature space (as in logistic regression). Its importance is that it is *composable*: by changing how units are connected, what nonlinearities they apply, and what signals drive learning, we can move from a single linear boundary to richer nonlinear decision surfaces when the task demands it. Many of these design choices are loosely inspired by how we think biological systems process information (as sketched in the section that follows), but the target remains an engineering model, not a faithful brain simulation.

How the rest of Part II builds.

- **Units → multilayer systems:** Chapter 6 chains units into multilayer networks; Chapter 7 shows how to compute all gradients efficiently (the training engine).
- **Architectures as inductive bias:** later chapters introduce structured connectivity (e.g., convolution, recurrence, attention) that changes what patterns are easy to represent and what information can flow.
- **Audit hooks:** keep learning curves, calibration, and slice checks alongside accuracy. A good loss is not automatically a good decision.

5.3 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 element-wise.

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 $\mathbf{A}^{(l)} \in \mathbb{R}^{B\times n_l}$, and weights map features by right multiplication $\mathbf{W}^{(l)} \in \mathbb{R}^{n_{l-1}\times n_l}$. Biases $\mathbf{b}^{(l)} \in \mathbb{R}^{n_l}$ broadcast across the batch: $\mathbf{Z}^{(l)} = \mathbf{A}^{(l-1)}\mathbf{W}^{(l)} + \mathbf{1}(\mathbf{b}^{(l)})^\top$. For a concrete check, if $B = 4$, $n_{l-1} = 3$, and $n_l = 5$, then $\mathbf{A}^{(l-1)} \in \mathbb{R}^{4\times 3}$, $\mathbf{W}^{(l)} \in \mathbb{R}^{3\times 5}$, and $\mathbf{Z}^{(l)} \in \mathbb{R}^{4\times 5}$. 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.

5.4 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 au-

thors 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.

5.5 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.

5.6 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.

5.7 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.

5.8 McCulloch-Pitts neuron: examples and limits

Recall the MP neuron definition in (5.3)–(5.4) (equivalently (5.6)); here we focus on logic-gate examples and the limitations that motivate learnable variants.

Example: AND and OR gates

- **AND:** For inputs x_1, x_2, set $w_1 = w_2 = 1$ and $\theta = 2$. The unit outputs 1 only when both inputs are 1, matching the AND truth table.
- **OR:** Keep $w_1 = w_2 = 1$ but set $\theta = 1$. The unit outputs 1 when at least one input is 1, matching OR.

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.

5.9 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

$$y = \begin{cases} 1 & \text{if } \mathbf{w}^\top \mathbf{x} + b \geq 0, \\ 0 & \text{otherwise,} \end{cases} \tag{5.8}$$

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. The signed-margin derivation below yields the update used in practice; 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.

Worked example: learning an OR gate by mistake-driven updates

We use labels $y \in \{-1, +1\}$ (so OR has targets $(-1, +1, +1, +1)$ for $(0,0), (0,1), (1,0), (1,1)$), learning rate $\eta = 1$, and initialize $\mathbf{w} = (0,0)$, $b = 0$. Cycle through the four inputs in that fixed order; whenever $y(\mathbf{w}^\top \mathbf{x} + b) \le 0$, apply $\mathbf{w} \leftarrow \mathbf{w} + y\mathbf{x}$, $b \leftarrow b + y$.

The sequence of updates (after each mistake) is:

step	(x_1, x_2), y	(w_1, w_2)	b
1	$(0,0), -1$	$(0,0)$	-1
2	$(0,1), +1$	$(0,1)$	0
3	$(1,0), +1$	$(1,1)$	1
4	$(0,0), -1$	$(1,1)$	0
5	$(0,0), -1$	$(1,1)$	-1
6	$(0,1), +1$	$(1,2)$	0
7	$(0,0), -1$	$(1,2)$	-1
8	$(1,0), +1$	$(2,2)$	0
9	$(0,0), -1$	$(2,2)$	-1

After these 9 updates, predicting $+1$ when $\mathbf{w}^\top \mathbf{x} + b \ge 0$ (and -1 otherwise) with $\mathbf{w} = (2,2)$, $b = -1$ labels all four OR inputs correctly.

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.

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.

Geometry intuition: beyond XOR (two triangles)

XOR is the canonical non-separable toy, but the limitation is geometric, not specific to a truth table. A second picture to keep in mind is *two interleaved triangles*: one class occupies the vertices of a large triangle, while the other occupies the vertices of a smaller triangle rotated inside it. No single line separates the two sets; any separating boundary must "bend" or be assembled from multiple linear pieces.

Why this matters. A perceptron draws exactly one hyperplane, so it cannot represent such boundaries. Chapter 6 restores expressivity by composing units so the overall decision boundary can be piecewise-linear (or curved) while still being trainable by gradients.

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

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

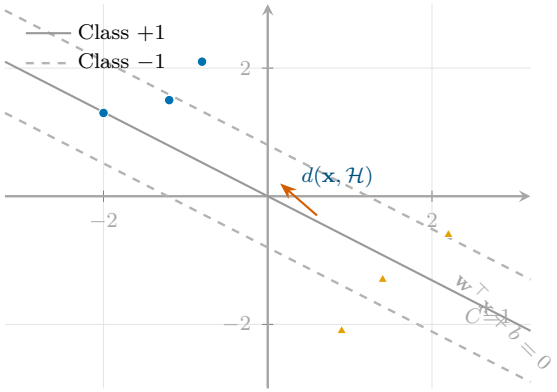

Figure 21: Perceptron geometry. Points on opposite sides of the separating hyperplane receive different labels, and signed distance to the boundary controls both prediction and update magnitude. Compare with Figure 17 in Chapter 4: both are linear separators, but logistic smooths the boundary into calibrated probabilities.

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, and it is also the source of the limitation.

Many real problems need *communities* of units that can specialize. Different hidden units respond to different regions, features, or patterns, and the model combines those responses into a richer decision surface. Multi-layer networks do not only add parameters; they add internal structure that lets different parts of the model "care about" different parts of the data.

Key takeaways

Minimum viable mastery

- 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).

Common pitfalls

- Expecting a single hyperplane to solve nonconvex structure: without hidden units you cannot express XOR-like logic.
- Mixing label codings ($\{-1, +1\}$ vs. $\{0, 1\}$) without adjusting the loss/update formulas.
- Treating linear scores as probabilities: calibrated probabilities require a probabilistic model/loss (e.g., logistic).

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.

If you are skipping ahead. Remember the two bottlenecks that force multilayer networks: expressivity (nonlinear boundaries) and trainability (smooth gradients). Chapter 6 and Chapter 7 assume these motivations.

Where we head next. Perceptrons are intentionally simple: hard thresholds and uniform updates. Their strengths (linear separation) and limits (for example XOR) motivate multilayer models. Chapter 6 continues this thread by chaining units, defining a loss, and asking the key training question: *how should weights change to improve performance?* That question leads directly to the chain rule and smooth activations. Chapter 7 then scales the same cache-and-chain-rule accounting to arbitrary depth and efficient implementation.

6 Multi-Layer Perceptrons: Challenges and Foundations

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.

We now take the smallest step beyond a single unit. We connect two neurons in series, treat the whole diagram as one trainable model, and choose a simple performance function that tells us when the output is moving in the right direction. This is the step where the simple unit becomes itself a building block. The central question is practical: *if performance improves when the output moves one way, how should each weight move?*

As soon as we try to compute those derivatives, a key obstacle appears: hard thresholds do not carry useful gradient information. Replacing them with smooth activations fixes the learning signal and gives us a clean update story. We keep the network small enough that every intermediate quantity is visible and derive the update rules once in full; scaling the same logic to deeper models is then a matter of organization.

Learning Outcomes

By the end of this chapter, you should be able to:

- Write a complete forward pass for a two-neuron chain, including intermediate quantities (p_1, y_1, p_2, y_2).
- Choose a simple scalar performance function P and explain what "improving performance" means in that setting.
- Use the chain rule to derive the gradients for $w_2, b_2, \mathbf{w}_1, b_1$ and interpret the δ notation as reusable local error signals.
- Explain (concretely) why hard thresholds break derivative-based updates and why smooth activations fix the learning signal.
- Sanity-check an update with one tiny numerical example before you scale the same logic to many layers.

Design motif

Start small. Keep every intermediate quantity visible. Let the chain rule explain how the learning signal moves, and use quick numerical checks to catch sign and accounting mistakes early.

6.1 From a single unit to the smallest network

A short map: building a trainable network. The chapter follows one tight loop:

- **Build:** write the forward computation (a two-neuron chain).
- **Judge:** define a scalar performance function P (we use squared error).
- **Move:** update parameters using the direction given by derivatives (gradient descent).

- **Fix:** Take out the threshold and make the unit differentiable so the learning signal can pass through it.

Once you can execute this loop for two neurons, scaling to many neurons is mostly accounting; we return to the organization when we scale up.

How this chapter fits the workflow. The objective-and-audit loop still applies; what changes is the *representation*. Once a model has internal computations, you have to keep track of intermediate variables and make sure the learning signal can actually pass through them. Before you scale up, do a few small sanity checks: confirm the forward pass, confirm the gradient signs, and check one tiny numerical update.

- 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 multi-layer network and derive its update rules, so the mechanics feel concrete.

Function estimation as the unifying view. Learning is about building a usable input→output mapping from examples. The difference from linear models is not the goal, but the *representation*: a neural network builds the mapping by composing simple units, so intermediate signals become part of the model. Each unit implements its own local mapping (from its input to its pre-activation and activation), and these local maps accumulate into a successful nonlinear mapping $f : X \to Y$. In this chapter we keep the accounting minimal—one tiny network and a simple squared-error objective—so we can focus on how the update rules emerge.

From one unit to a chain of units. A single unit (the perceptron) computes a weighted sum and then applies an activation:

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

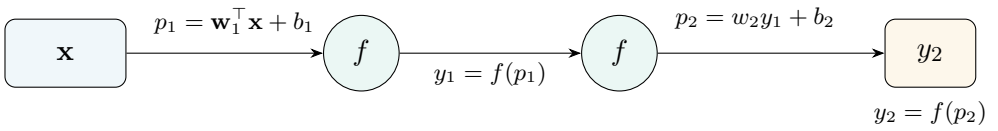

Figure 22: The minimal neural network used in this chapter is a two-neuron chain. The first unit produces an intermediate signal, and the second unit maps that signal to the final output.

where $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{w} \in \mathbb{R}^n$, and b is a bias. With one unit, the decision boundary $\mathbf{w}^\top \mathbf{x} + b = 0$ is a hyperplane, so a single unit can only produce linear separations; XOR is the canonical reminder of that limitation.

The smallest step beyond a single unit is a *two-neuron chain*: one unit feeds another. Write

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

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

This is the point where the unit becomes a building block: an intermediate signal y_1 is computed, reused, and then transformed again. Complexity now comes not only from what each block does in isolation, but from how the blocks connect, and how a change in one propagates to the other. We will reuse Figure 22 as our tracking diagram when we apply the chain rule.

Author's note: why a network changes the story. With one perceptron, every example pushes on the same single separator. But if you add a second unit, you are no longer forcing the entire problem through one shared knife-edge: you create intermediate signals that can specialize. Some units become sensitive to one pattern, others to another, and a later unit learns how to combine them.

I like to read those intermediate signals as learned transformations whose job is simple: make the final separation easier than it looked in the raw input space.

A checklist of what we must settle (and why). A diagram becomes a learning machine only after we make four choices explicit:

- **Parameters:** which numbers we are allowed to change (weights and biases).

- **Performance:** a single scalar P that measures how wrong the current output is.
- **Update:** a rule that changes parameters to reduce P.
- **Differentiability:** if the update is derivative-based, the path from parameters to P must be differentiable so the chain rule can assign credit (and blame).

We keep the network tiny so you can see all four ingredients at the same time.

Bias as a learned threshold. A threshold can be written as a bias. If $p = \mathbf{w}^\top \mathbf{x} - \theta$, define $b = -\theta$ and write $p = \mathbf{w}^\top \mathbf{x} + b$. In practice we append $x_0 = 1$ and treat b as an extra weight w_0; the algebra is identical. Intuitively, the bias controls *where* the unit switches, while the weights control *which directions* in input space push the unit toward switching. This simple trick lets us treat the bias exactly like another weight in both notation and updates.

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, penalizes large mistakes more than small ones, and keeps P smooth. That smoothness matters: if P changes continuously with the weights, derivatives can give a reliable direction for improvement. It is also easy to interpret and explain.

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

I start with squared error for one reason: it keeps the algebra short, so the chain rule is the only moving part you have to watch. The lesson here is not "squared error is always right." The lesson is learning how to take a forward computation and turn it into correct weight updates.
Once that story is clear, you can swap the objective and redo only the small set of derivatives that touch the output layer.

A geometric intuition. For a fixed input, the performance becomes a surface over the weights. If the surface is bowl-shaped, the best parameters sit near the bottom. Learning is then "navigation": move the weights in a direction that reduces performance. This is why it is preferable to use an algorithm that follows the local slope rather than guessing directions coordinate by coordinate. Figure 23 visualizes why moving in one vector direction is typically more efficient than changing one coordinate at a time.

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.

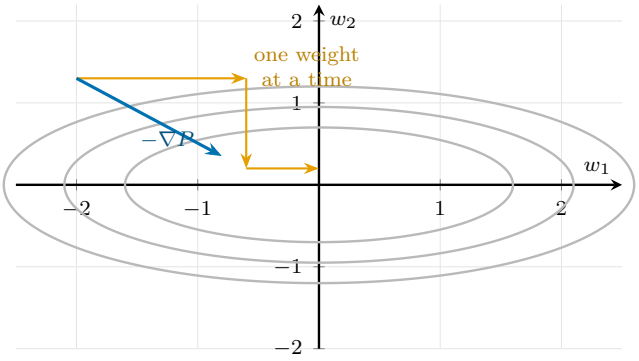

Figure 23: Think of performance as a surface over the weights. Gradient descent moves in one vector step (blue), whereas coordinate-wise updates can zig-zag (orange).

Step size is a design choice. The gradient gives a direction; the step size η sets how far you move. Too large and you bounce around (or diverge); too small and learning crawls. Many practical tricks are really ways of managing this tradeoff while keeping updates stable. In more uneven landscapes with multiple valleys, step size also affects *which* valley you settle into: aggressive steps can jump across good regions, while timid steps can get you stuck early.

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 the forward-pass equations in (6.3), factors like $f'(p_1)$ and $f'(p_2)$ appear immediately. Figure 24 makes this derivative contrast concrete.

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. Folding the threshold into a bias simplifies notation, but it does not remove the real issue: a hard step is still discontinuous. When you try to compute how a weight changes performance, the chain rule

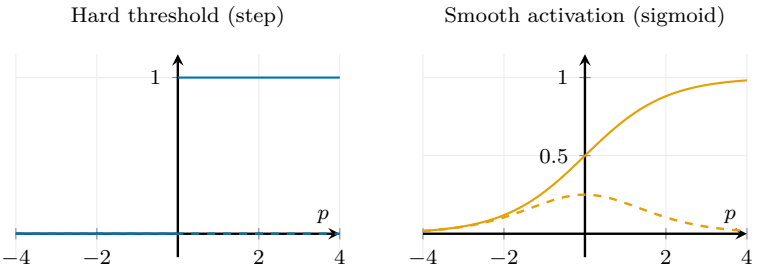

Figure 24: Hard thresholds block gradient-based learning because the derivative is zero almost everywhere. A smooth activation like the sigmoid provides informative derivatives across a wide range of inputs.

immediately introduces a derivative of the activation—terms of the form $f'(p)$ appear. With a step function, that derivative is either undefined (at the threshold) or zero almost everywhere, so the learning signal cannot pass through the unit in a useful way. This is why we replace the step with a smooth activation.

6.5 Differentiable activations and the sigmoid trick

A classic choice is the logistic (sigmoid) function

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

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 do not want to recompute expensive expressions during learning. With a sigmoid, once you have the forward output $y = \sigma(p)$, you also have its slope for free: $\sigma'(p) = y(1 - y)$.
That tiny identity hints at the bigger pattern: the forward pass is not only for predictions. It is also where you produce (and store) the intermediate values the backward pass will reuse. This idea, as intuitive as it might feel, is one of the reasons multi-layer perceptrons became trainable at scale and why modern deep learning can exist as an engineering discipline.

Derivation sketch. Let $\beta = \sigma(\alpha) = (1 + e^{-\alpha})^{-1}$. Differentiate and rewrite the result in terms of β:

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

The key point is that once you have computed β on the forward pass, you can compute the slope from β alone, without recomputing exponentials.

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 from the forward-pass equations in (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 $\dfrac{\partial p_2}{\partial y_1} = w_2.$
- $\dfrac{\partial p_1}{\partial \mathbf{w}_1} = \mathbf{x}.$

For the second-layer parameters, the chain rule gives

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

and similarly

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

For the first-layer parameters, the effect of the second layer appears explicitly 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

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

To make the reuse explicit, define local error terms:

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

Then $\partial P/\partial w_2 = \delta_2 y_1$ and $\partial P/\partial b_2 = \delta_2$. The first layer receives an error signal propagated backward:

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

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 what makes multi-layer training efficient and is why deeper networks remain tractable.

Worked example: one numerical gradient step (sanity check)

Take a single input $\mathbf{x} = [1, -1]^\top$, 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 $\nabla_{\mathbf{w}_1} P = \delta_1 \mathbf{x} \approx [-0.018,\ +0.018]^\top$ 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 $\mathbf{W}^{(l+1)}$, the error signal satisfies

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

where $\circ$ denotes element-wise multiplication. This recursion is the basic backward update pattern in multi-layer networks; the only real challenge is organizing the computation so you can evaluate it efficiently.

Author's note: what changes when you scale up

Going from two neurons to many layers does not change the idea. It is still the chain rule and the same local derivatives. What changes is the *organization*.

You run a forward pass that stores what you will need, then a backward pass that reuses those stored values to compute every gradient efficiently over a whole batch. Later we turn that into an implementable algorithm and show the accounting that keeps it reliable.

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 backward-recursion pattern used in deep networks.

Derivation closure: implement, cache, fail-fast

- **Implement:** treat the chapter equations as a forward function plus a scalar loss; write them once in vector form before batching.
- **Cache:** keep $(\mathbf{x}, p_1, y_1, p_2, y_2)$ from the forward pass so each gradient term is a local reuse, not a re-derivation.
- **Fail-fast checks:** run finite-difference gradient checks on a tiny example, then track train/validation divergence to catch update-sign or step-size mistakes early.
- **Handoff:** the same cache-then-backward discipline scales directly; only the index/shape accounting grows.

Key takeaways

If you remember only a few things. The summary in Section 6.8 is the core loop: a tiny computation graph, a scalar loss, and gradients that tell you how to move weights. The pitfalls below are the places people usually lose a sign, a factor, or a cached value. **Common pitfalls**

- Treating a step function like a differentiable activation and then trying to "repair" gradients by hand.
- Dropping a cached intermediate (or reusing the wrong one) and silently breaking the chain rule.
- Mixing up pre-activations and activations; when you batch, also keep shapes explicit so you do not rely on accidental broadcasting.

Exercises and lab ideas

Try these while the network is still small. These reinforce the two-neuron derivation and prepare you for multi-layer accounting.

- **Numerical gradient check:** Implement finite differences for the two-neuron chain and compare to your analytic gradients; report relative error and track down the first mismatch.
- **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 multi-layer perceptron (MLP) on XOR and plot its decision regions; note sensitivity to initialization and step size.

If you are reading for implementation. Be able to read a forward graph and a backward recursion: local derivatives, cached activations, and a scalar loss driving parameter updates. That is exactly what gets scaled up in deeper networks.

Where we head next. Chapter 7 answers the practical scaling question: how to go from the smallest network in this chapter to networks with many layers and

many units per layer. It uses the same ideas, but organizes them into a general recipe—a forward pass that caches intermediate quantities, and a backward pass that reuses those caches to compute gradients efficiently for deep, batched models.

7 Backpropagation Learning in Multi-Layer Perceptrons

Building on the two-neuron derivation in Chapter 6, we scale the same idea to an L-layer multi-layer perceptron (MLP). Conceptually it is still the chain rule; what changes is the accounting. We compute layerwise error signals (the δ's), reuse them to obtain every weight and bias gradient, and organize the work into a backward sweep you can implement and debug. The goal is practical: you should be able to trace one forward pass, write down one backward pass, and know exactly which stored intermediate values each gradient depends on. By the end of this chapter, the backpropagation algorithm should feel like a concrete procedure rather than a black box.

Learning Outcomes

- Derive the layerwise $\boldsymbol{\delta}$ recursion for an L-layer MLP and use it to write gradients for $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$.
- Keep a consistent "shape ledger" for activations, pre-activations, and error signals so vectorized code matches the algebra.
- Run one tiny numeric trace and one finite-difference gradient check to confirm your implementation before scaling up.
- Recognize the two common output-layer patterns: squared error (general) and softmax + cross-entropy (simplified δ).

Design motif

Treat the network as a computation you can trace. Cache what the forward pass computes, then sweep backward once and reuse the same local error signals to update every parameter.

7.1 Context and Motivation

Multi-layer networks raise a specific question: *How do we update the weights across multiple layers when the only explicit error signal is at the output?* In a single-layer perceptron, the output error touches the weights directly; in a deep network, a change in one layer propagates through subsequent layers and alters the output in a nonlinear, intertwined way.

Shallow networks (one hidden layer) already move beyond linear separability, but more complex tasks demand deeper hierarchies of features. The multi-layer perceptron stacks these layers to learn richer decision boundaries, and backpropagation is the mechanism that makes that depth trainable.

Intuition: backprop as a controlled ripple effect

Changing one early weight can only affect the loss by moving the intermediate quantities it feeds into. Backpropagation does not introduce a new idea; it is the chain rule written in an efficient order: run the forward pass once, cache the intermediates you will need, then flow sensitivities backward so each layer gets its own local error signal. Identities like $\sigma'(p) = y(1-y)$ (from Chapter 6) are valuable because they let you compute a derivative from cached forward values.

Implementation lens. A practical MLP rarely updates one coordinate at a time; gradients are treated as full vectors so every weight moves coherently. Backpropagation is what turns a multilayer diagram into a trainable model: it reuses local error signals so you can compute *all* gradients efficiently, making it practical to learn hidden representations (not just tune a final linear layer) while keeping the same validation→audit discipline (learning curves, early stopping, and slice checks—performance broken down by meaningful subgroups/conditions).

7.2 Problem Setup

Consider a multi-layer perceptron with layers indexed by $l = 0, 1, \ldots, L$, where $l = 0$ is the input layer and 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 algebra uncluttered (and aligned with Chapter 6), we use a simple squared-error objective. Let the network output be $\mathbf{a}^{(L)}$ and let $\mathbf{t}$ be the target. For classification, $\mathbf{t}$ is often one-hot (a vector with a 1 at the correct class index and 0 elsewhere). For a single example, a standard choice is the half-squared error:

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

It is often helpful to name the *error* explicitly. Define the componentwise output error as $e_k := a_k^{(L)} - t_k$ and the error vector $\mathbf{e} := \mathbf{a}^{(L)} - \mathbf{t}$. Then $\mathcal{L} = \frac{1}{2}\sum_k e_k^2 = \frac{1}{2}\|\mathbf{e}\|_2^2$, and the derivative you need to start the backward pass is $\partial\mathcal{L}/\partial a_k^{(L)} = e_k$ (squared error removes the sign inside $\mathcal{L}$, but the sign reappears in the derivative). When you train with a mini-batch, you typically use the *mean* loss over the batch; this introduces a factor $1/B$ in the gradients, which you can treat as part of the effective step size. 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.

Challenges in weight updates With more than one layer, the error you measure at the output does not tell each earlier weight what to do directly. Each weight affects the loss only through a chain of intermediate quantities: it shifts a pre-activation, changes an activation, and that change fans out through downstream units.

One way to phrase the bottleneck is *credit assignment*. If a small change in $w_{ij}^{(l)}$ would make the loss smaller, we want that weight to move in that direction; if it would make the loss larger, we want it to move the other way. The chain rule gives both the sign and the magnitude, but only if we compute the right intermediate sensitivities.

Backpropagation is the organization that makes those sensitivities reusable: it defines a local error signal (the δ's), computes them once from the output back toward the input, and then turns them into gradients for every weight and bias.

7.4 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.

Notation handoff. Across this chapter, a denotes activations and z denotes pre-activations; this pairing is reused in later deep-learning chapters. If you jump into chapters out of order, keep Appendix D nearby for symbol overloads.

7.5 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 targets t_k to form a loss (e.g., Equation (7.3)), and backpropagation propagates that output error backward through the layers to compute every weight gradient efficiently.

Mini example (aligned with this chapter): mean squared error (MSE) backprop on a two-layer MLP

```
# Shapes: X in R^{B x d}, W1 in R^{d x h}, W2 in R^{h x 1}
# Activations: H = f(Z1), y = f(Z2)  (use a smooth f)
def step_mse(X, t, params, eta):
    W1, b1, W2, b2 = params
    B = X.shape[0]
    # Forward pass
    Z1 = X @ W1 + b1
    H  = f(Z1)
    Z2 = H @ W2 + b2
    y  = f(Z2)
    # Loss (mean squared error over batch)
    P = 0.5 * ((y - t)**2).mean()
    # Backward pass (delta = dP/dZ)
    delta2 = (y - t) * fprime(Z2) / B
    grad_W2 = H.T @ delta2
    grad_b2 = delta2.sum(axis=0)
    delta1 = (delta2 @ W2.T) * fprime(Z1)
    grad_W1 = X.T @ delta1
    grad_b1 = delta1.sum(axis=0)
    # GD step
    return (W1 - eta * grad_W1, b1 - eta * grad_b1,
            W2 - eta * grad_W2, b2 - eta * grad_b2)
```

The elementwise product `*` mirrors the Hadamard (element-by-element) product notation from Equations (7.1) to (7.2). The key technical point is the `fprime` factors: this is exactly where hard thresholds break the learning signal.

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

If your gradients look "plausible" but training fails, check shapes first. For each layer:

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

Layers share this structure; convolutional/sequence models reuse the same calculus with different indexing and shape conventions.

7.6 Backpropagation: Recursive Computation of Error Terms

Our goal is the gradient of the loss with respect to each weight $w_{ij}^{(l)}$ connecting layer l to layer $l+1$, i.e., the weight from neuron i in layer l to neuron j in layer $l+1$.

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 overall structure are the same.

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

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

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 $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}}$ has the same shape as the weight matrix.

Interpretation. The term $\delta_j^{(l+1)}$ measures how sensitive the loss is to changes in the pre-activation $z_j^{(l+1)}$. Our task reduces to computing these δ terms for all neurons in the network.

7.6.1 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

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

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

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

where f' denotes the derivative of the activation function evaluated element-wise. With squared error, this is the entire story: $\partial \mathcal{L}/\partial a_k^{(L)} = a_k^{(L)} - t_k$, and the chain rule contributes the extra $f'(z_k^{(L)})$ factor.

For cross-entropy paired with a sigmoid or softmax output, the derivative of the log-likelihood removes that extra activation-derivative factor at the output, leaving $\delta_k^{(L)} = a_k^{(L)} - t_k$ (with δ still defined as $\partial \mathcal{L}/\partial z$). This simplification is one reason those loss/output pairs are so common: you avoid carrying an additional $f'(z)$ term in the output-layer learning signal.

7.6.2 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 $a_j^{(l)} = f\big(z_j^{(l)}\big)$, we have

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

Substituting into (7.14) yields

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

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

Summary: Backpropagation recursion Backpropagation is reverse-mode automatic differentiation (AD) on the network graph. A forward pass stores intermediate values (pre-activations and activations); a backward pass reuses those stored values to compute all gradients efficiently. Frameworks (PyTorch, JAX, TensorFlow) automate the mechanics, but the manual story is what lets you sanity-check shapes, signs, and scaling when training does something surprising.

The step-by-step output-layer derivation and the hidden-layer recursion are already established in Sections 7.6.1 and 7.6.2. The next step is to see how those same gradients are accumulated in batch and stochastic settings.

7.7 Batch and Stochastic Gradient Descent

Given a training set of N examples $\{(\mathbf{x}^{(n)}, \mathbf{t}^{(n)})\}_{n=1}^N$, 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 (shown here for a single output unit; the multi-layer case uses the same idea with layerwise δ's and cached activations):

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

- **Stochastic gradient descent (SGD):** Update weights after each training example using the instantaneous gradient $-\eta\, \delta^{(n)} \mathbf{x}^{(n)}$. The updates

are noisy, but that noise can be useful: it often keeps learning moving in flat regions and can help you avoid getting stuck too early.

Optimizer and stability notes

Start with plain SGD first; it makes it obvious whether the gradients you derived are doing the right thing. If training oscillates, add momentum; if step-size sensitivity dominates, try an adaptive method like AdamW (see Chapter 11). For regularization, tie this back to the L2 discussion in Chapter 3: in SGD, "weight decay" is weight shrinkage toward zero; in AdamW, that shrinkage is applied as a separate decay step rather than being mixed into the adaptive gradient. If gradients occasionally spike (common in deep or ill-conditioned nets), clipping is a practical safety valve. For classification, pair cross-entropy with a numerically stable softmax (log-sum-exp / subtract the max logit), as discussed in Chapter 11.

Implementation pattern (modern practice): softmax + cross-entropy

```
# Shapes: X in R^{B x d}, Y in R^{B x c} (one-hot)
#         W1 in R^{d x h}, W2 in R^{h x c}
# Note: stable softmax:
#   subtract row-wise max logit before exp.
def step_ce_softmax(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: for softmax + cross-entropy, delta2 = Yhat - Y
    delta2 = (Yhat - Y) / B
    grad_W2 = H1.T @ delta2 + wd * W2
    grad_b2 = delta2.sum(axis=0)
    delta1 = (delta2 @ W2.T) * relu_deriv(Z1)
    # Dropout backprop: reuse mask + scale
    delta1 = delta1 * mask1 / (1 - p_drop)
    grad_W1 = X.T @ delta1 + 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)
```

This snippet uses the same δ recursion as the squared-error derivation; the main change is the output layer, where softmax + cross-entropy yields $\delta^{(L)} = \hat{\mathbf{Y}} - \mathbf{Y}$ (up to batch scaling). Dropout and weight decay are optional here; they are shown only to indicate where they enter the forward and backward passes.

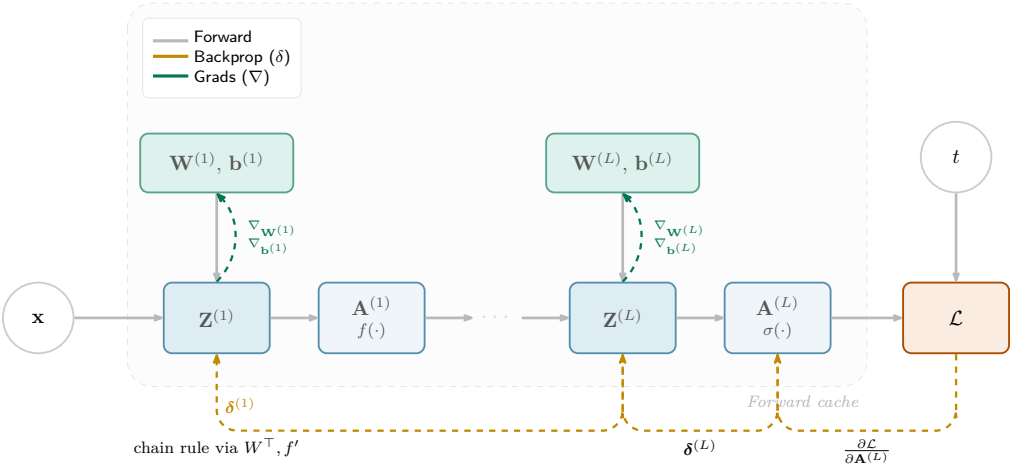

Figure 25: Computational graph for a feedforward network. Backpropagation is reverse-mode automatic differentiation (AD): the forward sweep caches intermediate values, and the reverse sweep propagates deltas while accumulating weight/bias gradients from those cached values.

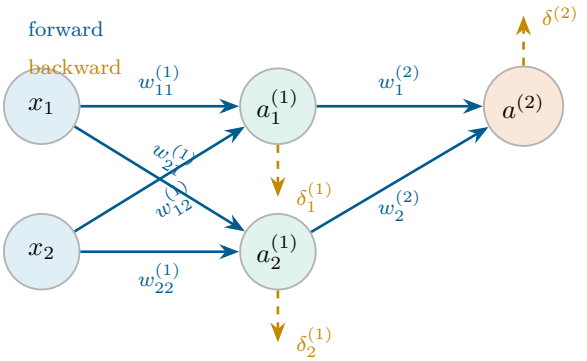

Figure 26: Forward (blue) and backward (orange) flows for a two-layer MLP. Cached activations and layerwise deltas travel along these arrows; backward signals use next-layer weights and activation derivatives.

Debugging and gradient-check checklist

- **Overfit a tiny batch:** Drive the loss near zero on a handful of samples; if you cannot, the issue is almost always code or setup, not generalization.
- **Finite differences on a tiny net:** Fix seeds, perturb one parameter, and compare analytic vs. numerical gradients. Do this before running long experiments.
- **Track per-layer gradient norms:** $\|\nabla \mathbf{W}^{(l)}\|$ that vanish (all zeros) or explode are early warnings.
- **Assert shapes and broadcasts:** Verify $\mathbf{Z}^{(l)}, \mathbf{A}^{(l)}, \boldsymbol{\delta}^{(l)}$ shapes and bias broadcasting explicitly.
- **Sanity baselines:** For a one-layer linear model, backprop should match the closed-form gradient you already know.

7.8 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

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

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

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

and

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

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.

For one output unit with activation $a = \sigma(z)$ and target t, define the scalar error $e = a - t$ and squared error $\mathcal{L}_{\text{SE}} = \frac{1}{2}e^2$. The output-layer error signal (still defined as $\delta = \partial\mathcal{L}/\partial z$) is

$$\delta^{(L)} = (a - t)\,\sigma'(z), \qquad \sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big).$$

For a hidden unit j in layer l, the same chain-rule logic gives

$$\delta_j^{(l)} = f'(z_j^{(l)}) \sum_k w_{jk}^{(l)}\, \delta_k^{(l+1)},$$

i.e., "local slope times the weighted sum of downstream error signals," with $w_{jk}^{(l)}$ the weight from unit j (layer l) to unit k (layer $l+1$).

Weight update rule (with momentum). Once you have δ's, turning them into weight updates is straightforward. For a weight $w_{ij}^{(l)}$ that connects unit i in layer $l-1$ to unit j in layer l, the gradient is proportional to "input times local error": $a_i^{(l-1)}\delta_j^{(l)}$. With momentum, one common update is

$$\Delta w_{ij}^{(l)}(n) = -\eta\, a_i^{(l-1)}(n)\, \delta_j^{(l)}(n) + \gamma \Delta w_{ij}^{(l)}(n-1), \tag{7.16}$$

followed by $w_{ij}^{(l)}(n) = w_{ij}^{(l)}(n-1) + \Delta w_{ij}^{(l)}(n)$. Here η is the step size and $\gamma \in [0, 1)$ is the momentum coefficient. (For the first hidden layer, $a_i^{(0)}$ is just the input feature x_i.) The index n denotes the update step (for example, the current training example in SGD, or the current mini-batch update).

A practical reading of η and γ. The learning rate sets the basic step scale; momentum averages recent gradient directions so updates do not zig-zag as much across narrow valleys. If training oscillates, reduce η; if it is painfully slow along a consistent direction, momentum can help.

One update step (what happens in one mini-batch)

1. **Initialize:** choose an initialization scale (Xavier is a common default for sigmoid/tanh; He is a common default for ReLU-family activations) and set biases (often zero).
2. **Forward pass:** compute and cache $z^{(l)}$ and $a^{(l)}$ for each layer.
3. **Loss + output signal:** compute $\mathcal{L}$ and $\delta^{(L)} = \partial\mathcal{L}/\partial z^{(L)}$.
4. **Backward pass:** propagate $\delta^{(l)}$ from $l = L-1$ down to the first hidden layer.
5. **Gradients:** form $\nabla_{\mathbf{W}^{(l)}}\mathcal{L}$ and $\nabla_{\mathbf{b}^{(l)}}\mathcal{L}$ from cached activations and δ's.
6. **Update:** apply your optimizer step (SGD, momentum, Adam) and repeat for the next mini-batch.

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. If you use dropout in a hidden layer, draw a 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 $\nabla_{\mathbf{W}^{(\ell)}}\mathcal{L}$ using cached activations (and cached dropout masks) so dropped units contribute zero gradient.
3. **Update block (per layer):**

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

Biases skip the weight-decay term. With Adam or SGD+momentum, $\mathbf{g}_\ell$ is what you feed into the optimizer update so regularization stays explicit.

Remarks Track both training and validation curves: a flat training loss usually points to an optimization issue (step size, activation saturation, initialization), while a widening train/validation gap points to capacity or regularization. Shuffle each epoch, checkpoint the best validation model, and let an early-stopping rule end runs before you overfit. In essence, the objective is to build confidence that your design decisions generalize out-of-sample; these curves give you a simple way to describe that generalization quantitatively.

7.9 Training Procedure and Epochs in Multi-Layer Perceptrons

An epoch is one complete pass over the training set. Most implementations shuffle the data and iterate over mini-batches: each update is cheaper than a full-batch gradient, and the resulting noise is often helpful for making progress.

1. Shuffle the training examples (or shuffle within class strata for imbalanced problems).
2. For each mini-batch: run a forward pass, compute the loss, run a backward pass, and update parameters.
3. At the end of the epoch: evaluate on a validation split, checkpoint the best model, and apply an early-stopping rule if needed.

After each epoch, look at both training and validation curves. A training loss that keeps falling while validation stalls is your cue to regularize, stop, or change capacity.

Common habits that prevent wasted runs.

- Start by overfitting a tiny batch and running a gradient check (the checklist box earlier in the chapter).
- Log the random seed and the full optimizer recipe; without it, "it did not train" is not a reproducible diagnosis.
- If you change one thing (activation, initialization, step size), keep everything else fixed so you can attribute cause and effect.

7.10 Role and Design of Hidden Layers

Hidden layers are where an MLP earns its flexibility. They do not just add parameters; they add intermediate representations, so different units can specialize and a later layer can combine those specializations. A *hidden layer* is any layer whose activations are not directly compared to the target: it sits between the input and output, and its outputs are internal signals the network learns to shape into useful features.

Design questions you actually have to answer.

- **Depth vs. width:** do you want a few wide layers or many narrow layers?
- **Capacity vs. data:** how much flexibility can your dataset support before you overfit?
- **Activation choice:** will gradients flow (ReLU family), or will they saturate (sigmoid/tanh) for the scale of your pre-activations? This connects directly to the smooth-activation discussion in Section 6.5.

A simple, defensible starting point.

- Pick one or two hidden layers and a moderate width, then let validation performance tell you whether you need more capacity.
- Choose an activation that keeps gradients alive (ReLU is a standard choice; if you see many dead units, a leaky ReLU is a simple alternative).
- Use early stopping and weight decay as your first line of defense against overfitting.

Trade-offs to keep in mind.

- **Too much capacity:** training loss drops quickly, validation does not; the model memorizes details you did not intend it to learn.
- **Too little capacity:** both training and validation plateau early; the model cannot represent the function you are asking for.

Example Execution Before you scale this to larger models, run one tiny network end-to-end and check each ingredient: compute the forward values, compute the output δ, push it one layer back, and confirm one gradient numerically. The explicit numbers in Section 7.8 can serve as a reference trace when you debug your own implementation.

Remarks on Convergence and Practical Considerations Backpropagation gives the *right* gradients; whether those gradients turn into learning depends on a handful of design and optimization choices. When training stalls or becomes unstable, the usual levers are:

- **Initialization scale:** keep pre-activations in a regime where derivatives are not all near zero (or wildly large).
- **Activation choice:** saturation (sigmoid/tanh) and dead regions (ReLU) show up directly as weak gradients.
- **Step size and schedule:** too large diverges, too small crawls; schedules and momentum smooth the path.
- **Regularization:** weight decay and dropout trade training fit for generalization.
- **Optimizer details:** momentum/Adam change the effective step direction and can rescue poorly conditioned problems.

Later deep-learning chapters revisit these choices in larger architectures; for now, it helps to see the canonical activation functions side by side.

Comparing canonical nonlinearities With the backprop derivation in place, it helps to compare the standard nonlinearities side by side. Figure 27 overlays the step, sigmoid, tanh, and ReLU curves so saturation regions and derivative behavior are visually apparent in one view.

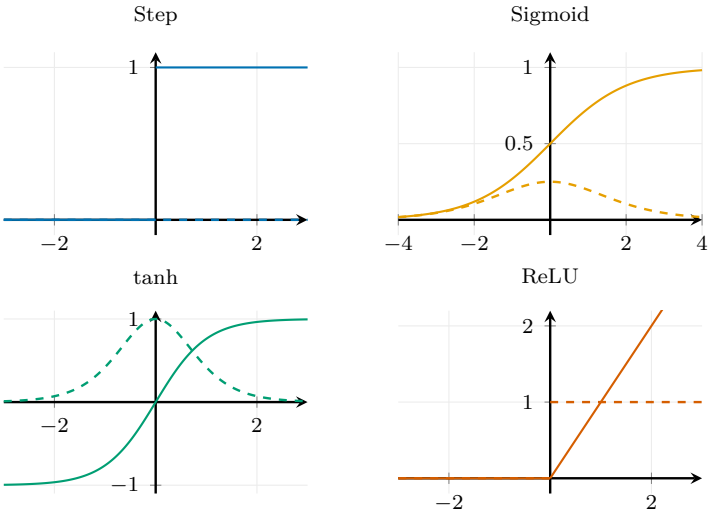

Figure 27: Canonical activation functions on a common axis. Solid curves show the activation; dashed curves show its derivative.

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 Some activations match the intuition of a smooth "firing curve"; others are chosen mainly because they train well. 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).

With these stability levers in mind, we now turn to a simple function-estimation case study.

7.11 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.

What to look for.

- **Capacity vs. fit:** if the model is too small, the best-fit curve will miss structure; if it is too large for your data density, it will fit noise or interpolation artifacts.
- **Activation effects:** saturated activations can make learning look "stuck" even when the model has enough parameters.
- **Data density:** uniform sampling over $[-3\pi, 3\pi]$ reveals whether your model interpolates smoothly between points or produces oscillatory artifacts.

Practical notes.

- This is a regression problem, not a classification problem.
- The target is nonlinear and oscillatory; underfitting and overfitting are both easy to see in a plot.
- If you report performance, report it as a reproducible experiment (seed, split, optimizer recipe). Avoid quoting a single "typical" number unless you can regenerate it.

7.12 Applications of Multi-Layer Perceptrons

MLPs show up in two roles. Sometimes they are the whole model, as in small function-approximation and classification tasks. More often they are trainable blocks inside larger systems: the same affine→nonlinearity pattern appears in convolutional networks, recurrent networks, and attention models. Later chapters return to this connection at scale. In all cases, the training signal still arrives as a loss at the output, and backpropagation is what assigns that loss to each parameter through the stored intermediate values from the forward pass.

7.13 Limitations of Multi-Layer Perceptrons

MLPs are flexible, but they require care to train reliably. The main limitations show up as training instability, sensitivity to design choices, and difficulty diagnosing silent implementation mistakes. Training can be sensitive: small changes to the random initialization, data order, or step-size settings can lead to noticeably different outcomes, especially when the problem is poorly conditioned.

- **Run-to-run variability:** different initializations and data orders can lead to different solutions. Treat seeds and splits as part of the experiment, not as incidental details.
- **Optimization fragility:** step size, activation choice, and initialization scale directly affect gradient flow. When learning stalls, use the gradient norms and the activation-derivative picture in Figure 27 as your first diagnostic.
- **Silent implementation bugs:** shape/broadcasting mistakes can produce gradients that look plausible but are wrong. This is why the shape ledger and finite-difference checks in this chapter are part of the basic implementation discipline.

7.14 Conclusion of Multi-Layer Perceptron Derivations

At this point you have the full backprop workflow: store computations from the forward pass, compute $\delta^{(L)}$, propagate δ's backward, and assemble gradients with the shapes kept honest. To close the derivation, it helps to write the same story once more in a compact matrix form.

Backpropagation algorithm recap (matrix form). One compact way to write the forward pass is:

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

where $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ are the weights and biases of layer l, $\mathbf{A}^{(l-1)}$ is the previous layer activation (rows are samples), $\mathbf{1} \in \mathbb{R}^B$ is an all-ones vector that broadcasts the bias across the batch, and $f^{(l)}$ is the (possibly layer-specific) activation function.

Define the error signal at layer l as:

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

Then the chain rule gives a reusable backward recursion:

$$\boldsymbol{\delta}^{(L)} = \nabla_{\mathbf{A}^{(L)}} \mathcal{L} \odot f^{(L)\prime}(\mathbf{Z}^{(L)}), \tag{7.17}$$

$$\boldsymbol{\delta}^{(l-1)} = \left(\boldsymbol{\delta}^{(l)} (\mathbf{W}^{(l)})^{\top}\right) \odot f^{(l-1)\prime}(\mathbf{Z}^{(l-1)}), \quad l = L, \ldots, 2, \tag{7.18}$$

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

Finally, turn those δ's into parameter gradients:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = (\mathbf{A}^{(l-1)})^{\top} \boldsymbol{\delta}^{(l)}, \tag{7.19}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} = \mathbf{1}^{\top} \boldsymbol{\delta}^{(l)}. \tag{7.20}$$

If your loss is defined as a mean over the batch, divide these batch sums by B; the shape ledger earlier in the chapter uses that mean convention.

These equations are the whole story: cache forward values, propagate δ's backward, then assemble gradients. Figure 26 complements the algebra by showing how cached activations (forward arrows) line up with backward error signals in a simple two-layer network.

Key takeaways

If you remember only a few things

- Backpropagation is the chain rule organized: cache forward values, propagate δ's backward using Equation (7.18), then assemble gradients with Equations (7.19) and (7.20).
- The δ's are not mysterious: they are sensitivities ($\partial \mathcal{L}/\partial \mathbf{Z}^{(l)}$) that you reuse to update every parameter.
- For softmax + cross-entropy, the output-layer error simplifies to $(\hat{\mathbf{Y}} - \mathbf{Y})/B$; a small finite-difference check can confirm this identity, and it is used in the modern-practice code pattern earlier in the chapter.
- When training misbehaves, debug like an engineer: overfit a tiny batch, run a finite-difference check, then inspect per-layer gradient norms and shapes.

Common pitfalls

- Dropping a transpose: $(\mathbf{W}^{(l)})^\top$ is easy to miss and produces gradients that look "reasonable" but train the wrong model.
- Mixing up pre-activations and activations: $f'(\mathbf{Z})$ depends on what you cache.
- Losing the batch scaling: decide whether you average over the batch ($/B$) and do it consistently.
- Silent broadcasting mistakes: bias terms and per-example deltas can accidentally align in NumPy/PyTorch and hide a bug.

Practical early stopping and checkpointing

- Keep a validation split separate from the training mini-batches, and record $L_{\text{val}}^{(e)}$ after each epoch.
- Stop when L_{val} has not improved for k consecutive epochs (a patience of $k \in [5, 10]$ is a reasonable first try). If the curve is noisy, require a small minimum improvement before you reset patience.
- Always checkpoint the parameters that achieved the best validation score and restore them before testing. If training is noisy, averaging the last few checkpoints can stabilize performance.

Derivation closure: implement, cache, fail-fast

- **Implement:** write one clean layer routine $(\mathbf{A}^{(l-1)}, \mathbf{W}^{(l)}, \mathbf{b}^{(l)}) \mapsto (\mathbf{Z}^{(l)}, \mathbf{A}^{(l)})$, then reuse it for every layer.
- **Cache:** store what backprop needs later (at minimum $\mathbf{A}^{(l-1)}$ and $\mathbf{Z}^{(l)}$). Do not recompute forward quantities during the backward pass.
- **Fail fast:** gradient-check one tiny network and overfit one tiny batch before you trust results on a real dataset.
- **Make runs comparable:** record seeds, optimizer, schedule, and stopping rule. The reporting checklist in Appendix E helps keep comparisons honest.

Exercises and lab ideas

- **A full trace on paper:** pick a tiny network and write out the forward values and δ's by hand, then match them against a short script.
- **Finite differences:** implement a gradient check for one layer and report the worst relative error; track down the first mismatch.
- **Activation choice:** run the same task with sigmoid/tanh/ReLU and compare gradient norms by layer; relate what you see to Figure 27.
- **Early stopping:** train until overfitting is visible, then add early stopping and report how the best checkpoint changes.

If you are jumping chapters. Keep one habit: run a tiny gradient check before scaling up.

Where we head next. Chapter 8 introduces radial basis function networks, an alternative nonlinear route where many hidden features are fixed and the output layer can be solved more directly. This provides a clean contrast to end-to-end backpropagation and a different training decomposition to compare against.

8 Radial Basis Function Networks (RBFNs)

Building on the multilayer perceptron (MLP) architecture (Chapter 6) and its training machinery (Chapter 7), this chapter introduces radial basis function networks (RBFNs): three-layer models with fixed nonlinear bases and a linear readout. Figure 1 places this as the kernel/prototype branch alongside the MLP path.

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.

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.

8.1 Overview and Motivation

We start with the three-layer picture and the basis-function intuition, then write the same story in matrix form so training becomes a regularized least-squares

solve. Near the end, we connect the finite-basis picture to the kernel viewpoint. (The short Wiener-filter box is optional context if you want the signal-processing parallel.)

Unlike MLPs, which learn weights in every layer, an RBFN separates the job into two parts. The hidden layer provides a fixed nonlinear feature map (set by centers and widths). The output layer then learns a linear combination of those features. That split is the main story here: choose the basis, then solve for the readout.

The key idea is that "nonlinear" can come from the representation rather than the readout. Radial basis functions act like localized, kernel-style features; once you lift inputs into that feature space, the output layer is still linear, and the weights are responsible for finding the separating border. This is closely related to the kernel trick used in SVMs; Appendix B collects the classical kernel viewpoint.

Chapter 3 frames this as a bias–variance tuning problem (capacity, regularization, and diagnostics via learning curves). Kernel methods such as kernel ridge regression and support vector machines (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.

Running example: XOR as a representation problem. XOR is the smallest reminder that a single line in the input space is not always enough. The point of RBF networks is not that the output layer becomes complicated; it stays linear. The complication is pushed into the feature map: the hidden units apply localized, kernel-like transforms, and the output weights are responsible for finding the separating border in that transformed space. We will return to XOR twice: first to see the feature map, then to see how the linear solve chooses the readout weights.

Centers from clustering (a practical default). 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.

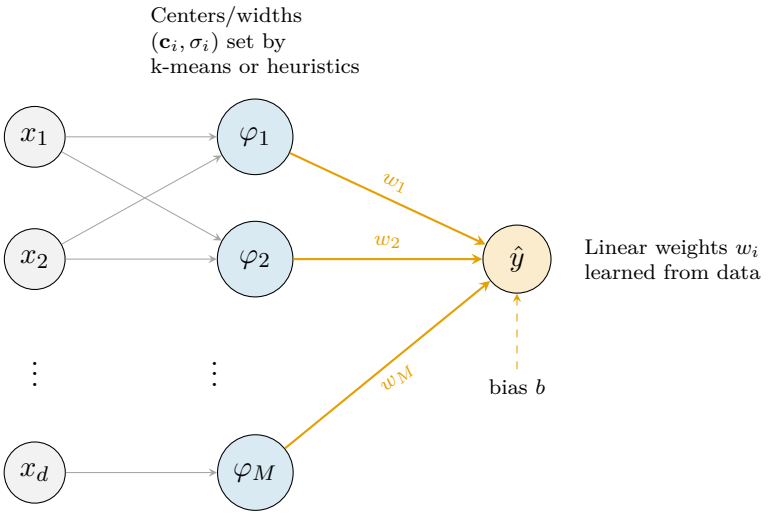

Figure 28: RBFN architecture. Inputs feed fixed radial units parameterized by centers and widths; a linear readout with weights and bias is trained by a regression or classification loss. Only the output weights are typically learned, while centers and widths come from clustering or spacing heuristics.

8.2 Architecture of RBFNs

The RBFN consists of three layers:

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 $\mathbf{\Phi} \in \mathbb{R}^{N \times M}$ stacks one transformed sample per row, with entries $\mathbf{\Phi}_{ji} = \varphi_i(\mathbf{x}_j) = G_i(\mathbf{x}_j)$. This matches the design-matrix convention used in Chapter 3.

Figure 28 highlights the split between fixed radial features and a trained linear readout.

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.

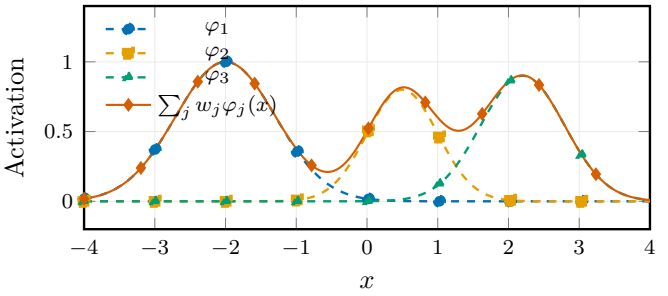

Figure 29: Localized Gaussian basis functions (dashed) and their weighted sum (solid). Overlapping bumps allow RBF networks to interpolate complex signals smoothly.

Figure 30 compares center-placement strategies used during initialization.

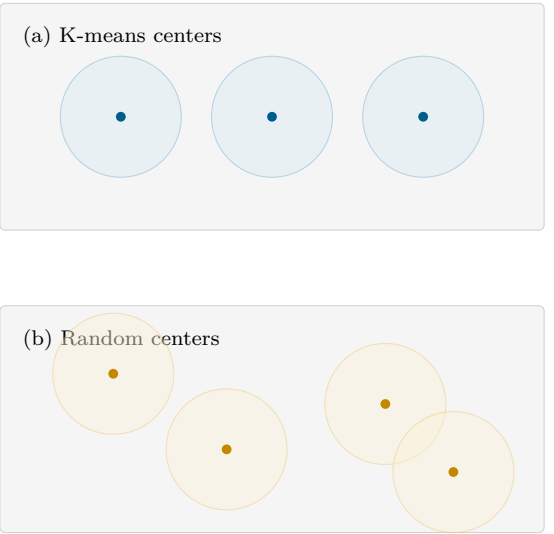

Figure 30: Center placement and overlap (schematic). K-means prototypes tend to tile the data manifold more evenly than random picks, giving more uniform receptive-field overlap. Random centers can leave gaps or create excessive overlap, which affects the width (sigma) choice and the conditioning of the later linear solve.

Later in the chapter we contrast this finite-basis ("primal") view with the kernel ("dual") view and show how Nyström-style approximations fit into the same story.

Figure 28 plus the overview above summarize the three-layer flow. Next we write the same story algebraically, so the training step becomes a linear solve with shapes you can check. 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.

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

Some datasets are not linearly separable in the original input space. A nonlinear feature map can move the same points into a space where a single linear boundary is enough.

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 (XOR).

- Inputs: $\mathbf{x} \in \{0,1\}^2$ with the four corners $(0,0), (0,1), (1,0), (1,1)$.
- Two radial units centered at $\mathbf{c}_1 = (0,0)^\top$ and $\mathbf{c}_2 = (1,1)^\top$.
- Feature map: $\mathbf{g}(\mathbf{x}) = [g_1(\mathbf{x}), g_2(\mathbf{x})]^\top$ with Gaussian g_i.
- Linear readout: $y = \mathbf{w}^\top \mathbf{g}(\mathbf{x})$.

Assumptions:

- For simplicity, set $\sigma^2 = 1$ (so $2\sigma^2 = 2$) in the Gaussian activation.
- Use two centers: $\mathbf{c}_1 = (0,0)^\top$ and $\mathbf{c}_2 = (1,1)^\top$.

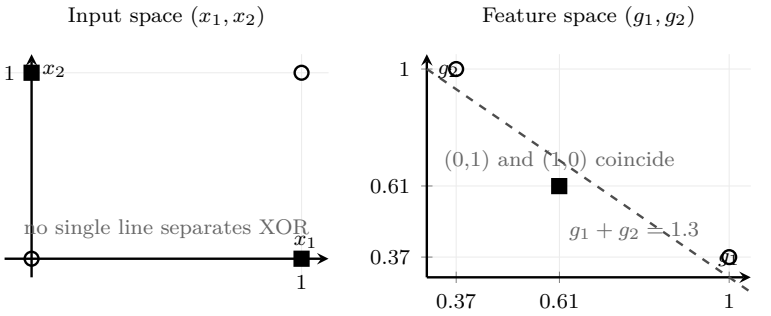

Figure 31: XOR before and after an RBF feature map. Left: in (x_1, x_2), no single line separates the labels. Right: in (g_1, g_2), the transformed points are linearly separable; one valid separating border is $g_1 + g_2 = 1.3$ (equivalently $w = [1, 1]$, $b = -1.3$).

- Use a Gaussian radial basis function (RBF):

$$g_i(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{c}_i\|^2}{2\sigma^2}\right).$$

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 $(e^{-1}, 1)$. The two off-diagonal corners $(0, 1)$ and $(1, 0)$ map to the same feature point $(e^{-1/2}, e^{-1/2})$. In this feature plane the XOR labels become linearly separable; for instance, the separator $g_1 + g_2 = 1.3$ places the diagonal corners on one side and the off-diagonal corners on the other (Figure 31).

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. Using the design matrix $\mathbf{\Phi}$ defined in the notation box, the model predicts $\hat{\mathbf{d}} = \mathbf{\Phi}\mathbf{w}$, and the least-squares objective is

$$J(\mathbf{w}) = \|\mathbf{d} - \mathbf{\Phi}\mathbf{w}\|^2. \tag{8.3}$$

Differentiating (8.3) with respect to $\mathbf{w}$ and setting the gradient to zero yields

$$\mathbf{\Phi}^\top \mathbf{\Phi}\, \mathbf{w} = \mathbf{\Phi}^\top \mathbf{d}. \tag{8.4}$$

When $\mathbf{\Phi}^\top \mathbf{\Phi}$ is well conditioned, the closed-form solution is $\mathbf{w}^\star = (\mathbf{\Phi}^\top \mathbf{\Phi})^{-1} \mathbf{\Phi}^\top \mathbf{d}$; 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, $\mathbf{\Phi}^\top \mathbf{\Phi}$ 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 nonlinear map $g(\cdot)$ and its role in constructing $\mathbf{\Phi}$ were defined in the transformation example above; here we focus on its parameters and how to choose it.

Two parameters characterize $g(\cdot)$:

- $\mathbf{c}_i$: the 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 $r = \|\mathbf{x} - \mathbf{c}\|$. 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 (the region of input space that activates the unit strongly) centered at $\mathbf{c}$ with width σ. It is the most commonly used kernel in RBF networks.

Why "radial"? 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 32.

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.

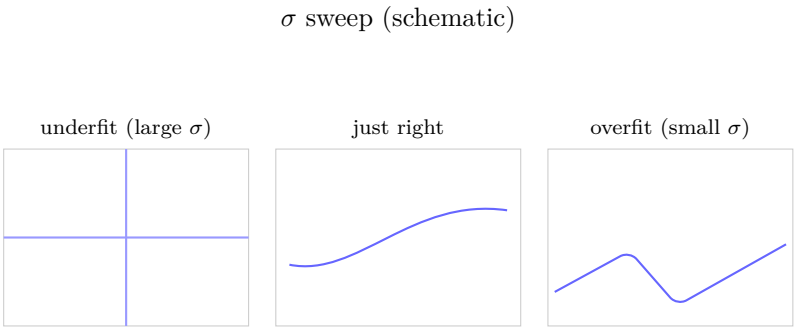

Figure 32: Schematic illustration of how the width parameter σ influences decision boundaries: too-large σ underfits (bases overlap heavily and wash out locality), intermediate σ captures the boundary, and too-small σ can produce fragmented regions. For a computed XOR example, see Figure 34.

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 activations

$$\varphi_i(x) = \exp\left(-\frac{(x - c_i)^2}{2\sigma^2}\right),$$

which serve as localized similarity measures to each center c_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 center elicit large activations, yielding sharply varying boundaries that can overfit noise. Visualizing $\varphi_i(x)$ for several centers illustrates how tuning σ controls the flexibility of the classifier.

We return to this tuning picture again later with an XOR-style toy example, where the decision boundary is easy to visualize.

Notation note. In this chapter we write radial basis functions as $\varphi_i(\cdot)$ and use $\mathbf{\Phi}$ 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 $\mathbf{c}_i$ with spread parameters σ_i. The output is a linear combination of these nonlinear transformations. The key challenge is to determine the parameters:

$$\{\mathbf{c}_i, \sigma_i, w_i\}_{i=1}^{M},$$

where M is the number of hidden neurons.

Finding the Centers $\mathbf{c}_i$: 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:

$$\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_M.$$

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 center $\mathbf{c}_i$ and its nearest neighboring centers. 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

$$\varphi_i(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{c}_i\|^2}{2\sigma_i^2}\right).$$

The training problem reduces to solving the linear system:

$$\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{\Phi}\mathbf{w}\|^2, \tag{8.5}$$

where $\mathbf{y}$ is the vector of target outputs and $\mathbf{\Phi}$ is the design matrix with entries $\mathbf{\Phi}_{ji} = \varphi_i(\mathbf{x}_j)$. When $\mathbf{\Phi}^\top\mathbf{\Phi}$ is well-conditioned, the ordinary least-squares solution is

$$\mathbf{w}^\star = (\mathbf{\Phi}^\top\mathbf{\Phi})^{-1}\mathbf{\Phi}^\top\mathbf{y}.$$

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

$$\min_{\boldsymbol{\alpha}} \|\mathbf{y} - \mathbf{K}\boldsymbol{\alpha}\|^2 + \lambda \boldsymbol{\alpha}^\top \mathbf{K}\boldsymbol{\alpha},$$

where $\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$ 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 a Nyström approximation: $\boldsymbol{\Phi}\mathbf{w}$ projects onto a subset of kernel features using $M < N$ landmark bases, rather than expanding on all N training points.
Numerically, $\boldsymbol{\Phi}^\top \boldsymbol{\Phi}$ can be ill-conditioned if the bases overlap excessively or if centers cluster tightly; kernel ridge has the same issue via $\mathbf{K}$.
Regularization is therefore essential: add λI before inversion,

$$\mathbf{w}^\star = (\boldsymbol{\Phi}^\top \boldsymbol{\Phi} + \lambda I)^{-1} \boldsymbol{\Phi}^\top \mathbf{y},$$

mirroring the $\lambda \boldsymbol{\alpha}^\top \mathbf{K}\boldsymbol{\alpha}$ 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 33 summarizes the primal and dual viewpoints side by side and highlights where a finite basis acts like a Nyström approximation.

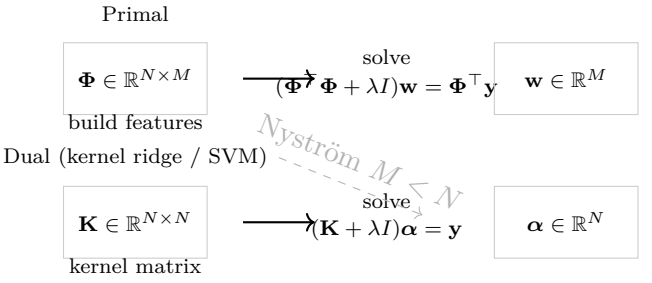

Figure 33: Primal (finite basis) vs. dual (kernel ridge) viewpoints. Using as many centers as data points recovers the dual form; using fewer centers corresponds to a Nyström approximation. The same trade-off appears in kernel methods through the choice of kernel and effective rank.

To improve numerical stability or control model complexity, a Tikhonov (ridge) regulariser can be added,

$$\mathbf{w}_\lambda^\star = (\mathbf{\Phi}^\top\mathbf{\Phi} + \lambda I)^{-1}\mathbf{\Phi}^\top\mathbf{y}, \qquad \lambda > 0,$$

or more generally one can use the Moore–Penrose pseudoinverse $\mathbf{\Phi}^+$ when $\mathbf{\Phi}^\top\mathbf{\Phi}$ is singular, yielding $\mathbf{w}^\star = \mathbf{\Phi}^+\mathbf{y}$. A quick dimensional sanity check is that $\mathbf{\Phi} \in \mathbb{R}^{N\times M}$, $\mathbf{w} \in \mathbb{R}^M$, 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 $\mathbf{c}_i$ (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 $\mathbf{\Phi}$ with entries $\mathbf{\Phi}_{ji} = \varphi_i(\mathbf{x}_j)$; choose a small grid of λ values and solve $(\mathbf{\Phi}^\top \mathbf{\Phi} + \lambda I)\mathbf{w} = \mathbf{\Phi}^\top \mathbf{y}$.
4. Evaluate on a validation set and pick (σ, λ, M) that minimizes validation loss; for classification, cross-entropy or hinge losses are also feasible with the same design matrix $\mathbf{\Phi}$.

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 $\mathbf{\Phi}$ with entries $\mathbf{\Phi}_{ji} = \exp\big(-\|\mathbf{x}_j - \mathbf{c}_i\|^2/(2\sigma^2)\big)$ and solve $(\mathbf{\Phi}^\top \mathbf{\Phi} + \lambda I)\mathbf{w} = \mathbf{\Phi}^\top \mathbf{t}$ over a small grid $\lambda \in \{10^{-4}, 10^{-3}, 10^{-2}\}$. The best λ yields a linear separator in the lifted $\mathbf{\Phi}$-space that classifies XOR. Widening σ makes bases overlap heavily and can wash out locality (very smooth boundaries and low margins), while shrinking σ makes the model extremely local and can

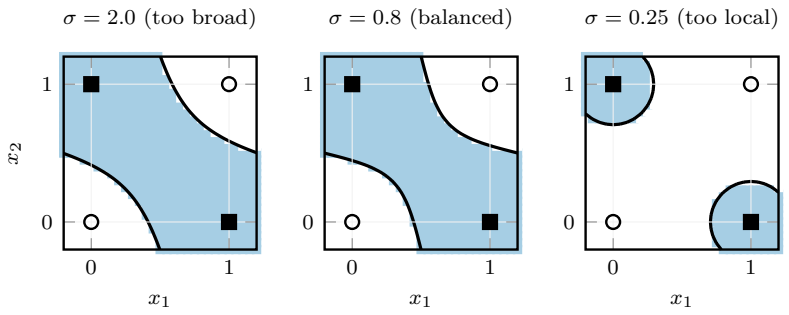

Figure 34: Effect of σ on an RBFN XOR boundary (4 centers at the data corners, ridge $\lambda = 10^{-3}$, threshold 0.5). Too-large σ makes bases overlap heavily, producing a very smooth, low-contrast boundary; intermediate σ yields a cleaner separation; too-small σ makes the model extremely local, producing small "islands" around prototypes.

create small islands that fit the training points but generalize poorly. Ridge helps whenever $\mathbf{\Phi}^\top \mathbf{\Phi}$ is poorly conditioned (typically when bases overlap heavily or centers cluster).

Figure 34 shows one such boundary for a fixed choice of centers, width, and ridge regularization.

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 $\mathbf{c}_i$, 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.

Sidebar (optional): Wiener filtering in one paragraph

The Wiener filter is the same least-squares projection story in signal-processing notation. In the Wiener case, you write the linear estimator as $y(t) = \mathbf{w}^\top \mathbf{x}(t)$ and the normal equations involve second-order statistics $\mathbf{R} = \mathbb{E}[\mathbf{x}\mathbf{x}^\top]$ and $\mathbf{p} = \mathbb{E}[d\,\mathbf{x}]$, giving $\mathbf{R}\mathbf{w}^\star = \mathbf{p}$. In an RBF network, you replace the raw input $\mathbf{x}$ by fixed nonlinear features $\mathbf{\Phi}$ (built from radial units), and the same idea becomes $(\mathbf{\Phi}^\top \mathbf{\Phi} + \lambda I)\mathbf{w}^\star = \mathbf{\Phi}^\top \mathbf{y}$. The point of the sidebar is just this mapping: both are "fit a linear readout to fixed features," and conditioning/regularization control how stable that fit is.

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

Minimum viable mastery

- 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.

Common pitfalls

- Width selection: too small memorizes, too large collapses to a linear model; validate σ and λ.
- Poor conditioning: without regularization, solves can be numerically unstable even when the math is correct.
- Confusing kernels with bases: a full kernel method scales differently than a truncated (primal) basis expansion.

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.

If you are skipping ahead. Keep the idea that "nonlinear" can still be linear-in-parameters once a basis is fixed. That perspective is reused in kernel methods and shows up again when we discuss architectural bias (CNNs) and representation choices.

Where we head next. Chapter 9 moves from supervised objectives to unlabeled competitive learning and prototype organization. Chapter 10 then revisits recurrence through an energy-based lens, setting up later sequence-model chapters.

9 Introduction to Self-Organizing Networks and Unsupervised Learning

Chapter 8 kept nonlinearity close to linear algebra: once you fix a set of basis functions, you lift the inputs into a feature space and (often) solve a regularized least-squares problem for the output weights. Now we switch the question. Instead of "how do I fit targets?", we ask: when labels are missing, expensive, or not even the right interface, what structure is already present in the inputs?

This chapter opens the unsupervised neural thread with *Self-Organizing Maps* (SOMs), also known as Kohonen maps. You learn a set of prototypes and place them on a grid so that, as training progresses, units that are neighbors on the grid tend to respond to similar inputs. Chapter 10 then studies Hopfield networks as an energy-based associative memory model. Both operate without explicit targets, but they use that freedom differently: SOMs emphasize organization and visualization, while Hopfield emphasizes retrieval dynamics.

Learning Outcomes

- Describe how competitive learning, cooperation, and annealing interact in SOM training.
- Monitor SOM quality via quantization error (QE), topographic error (TE), and interpret U-matrices (unified distance matrix plots).
- Connect SOMs to broader unsupervised techniques (clustering, dimensionality reduction) and know when to use each.

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.

9.1 Overview of Self-Organizing Networks

Self-organizing networks aim to discover structure in input data without labels. The most prominent example is the *Self-Organizing Map* (SOM), introduced by Teuvo Kohonen. SOMs are used for clustering and for building a visualization-friendly map of a high-dimensional dataset.

The idea is fairly intuitive. If the data have a meaningful notion of similarity in the original space, then repeated *competition* (pick the best-matching unit) and *cooperation* (update its neighbors as well) can organize a fixed 2D *grid* of prototype vectors so that nearby grid locations tend to represent nearby regions of the data. The map is not a perfect geometric embedding; it is an engineered bias that often produces a useful, inspectable picture.

SOMs are trained without labeled outputs; the only signal is input similarity under your chosen distance. Learning is *competitive* (each input picks a best-matching unit) but also *cooperative* (neighbors of the winner move too). That

neighborhood coupling is the deliberate bias: it encourages nearby grid locations to represent similar inputs, so the map can be read both as a set of prototypes and as a visualization.

Historical intuition: two sheets and topographic neighborhoods

A useful way to picture SOMs is as two coupled "sheets": an input space and a fixed 2D grid of units. Each input is connected (in principle) to the whole grid, but learning makes some regions 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 grid if they are statistically similar under the features the SOM has learned.

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 grid. 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 grid distance and shrinks as training progresses, so early updates promote global organization while later updates refine only the closest units. Figure 35 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.

Figure 44 pairs feature-plane views with U-Matrix diagnostics for SOM audits.

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*.

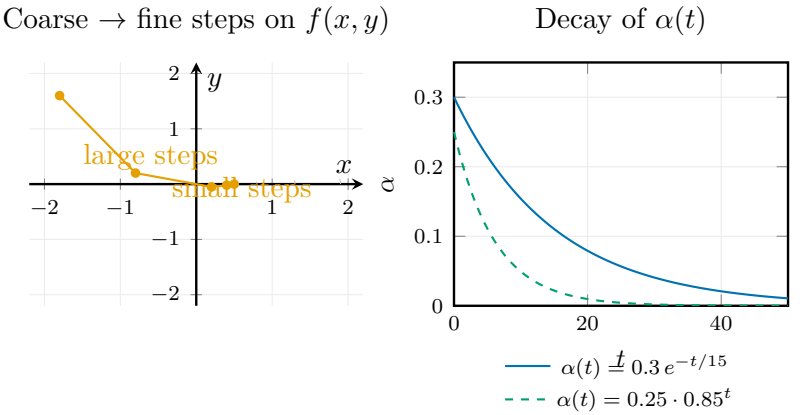

Figure 35: Learning-rate scheduling intuition (schematic). On a smooth objective (left), large initial steps quickly cover ground and roughly align prototypes, while a decaying step-size refines the solution near convergence. Right: common exponential and multiplicative decays used in SOM training.

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: Think of clustering as "discovering operating modes" from measurements. Your $\mathbf{x}_i$ could be a feature vector extracted from vibration spectra, network traffic, or sensor logs. Without labels, you still want the algorithm to separate one regime from another because the points inside a regime are genuinely similar in the feature space.

K-means Clustering: A classical and widely used clustering algorithm is *K-means*, which operates as follows:

1. Initialize K cluster centroids $\{\mathbf{m}_1, \mathbf{m}_2, \ldots, \mathbf{m}_K\}$ randomly.
2. For each data point $\mathbf{x}_i$, assign it to the cluster with the nearest centroid:

$$c_i = \arg\min_k \|\mathbf{x}_i - \mathbf{m}_k\|_2, \tag{9.1}$$

   where $\|\cdot\|_2$ denotes the Euclidean norm.
3. Update each centroid as the mean of all points assigned to it:

$$\mathbf{m}_k = \frac{1}{|C_k|} \sum_{\mathbf{x}_i \in C_k} \mathbf{x}_i, \tag{9.2}$$

   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.

Keep the K-means update in mind: SOMs reuse the same prototype-moving idea, but add a neighborhood on a fixed 2D grid so the prototypes are not only learned, but also *organized* for inspection.

9.3 Dimensionality Reduction: Simplifying High-Dimensional Data

Dimensionality reduction is what you do when the ambient dimension is too large to see and reason about directly. You accept some information loss, but you try to preserve the relationships you care about (variance, distances, neighborhoods) so the reduced view remains useful. This matters for:

- **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.

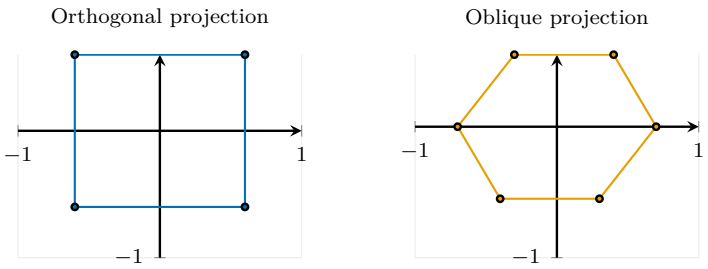

Figure 36: Classical MDS intuition (schematic). Projecting a cube onto a plane via an orthogonal map yields a square (left), whereas an oblique projection along a body diagonal produces a hexagon (right). The local adjacency of vertices is preserved even though metric structure is distorted.

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 36 illustrates these two views.

Common techniques include Principal Component Analysis (PCA), which preserves directions of maximum variance, and classical Multidimensional Scaling (MDS), which reconstructs a configuration whose pairwise distances match the original ones as closely as possible (via double-centering the squared-distance matrix and an eigen-decomposition). Nonlinear methods such as t-distributed stochastic neighbor embedding (t-SNE) or Uniform Manifold Approximation and Projection (UMAP) emphasize local neighborhoods but typically sacrifice global distance fidelity. SOMs will give us a different kind of reduction: a *discrete* map built from learned prototypes on a 2D grid, not a continuous coordinate chart.

9.4 Dimensionality Reduction and Feature Mapping

The dimensionality-reduction goals were covered in Section 9.3; here we make explicit what kind of "mapping" a SOM actually learns. A SOM does not return a continuous coordinate system like PCA or classical MDS. Instead, it learns a

finite set of prototype vectors $\{\mathbf{w}_i\}$ arranged on a low-dimensional grid with fixed coordinates $\{\mathbf{r}_i\}$. Each input $\mathbf{x} \in \mathbb{R}^n$ is mapped to the grid by choosing its best matching unit (BMU),

$$c(\mathbf{x}) = \operatorname*{argmin}_i \|\mathbf{x} - \mathbf{w}_i\|_2^2,$$

and then using $\mathbf{r}_{c(\mathbf{x})}$ (or a neighborhood-smoothed variant) as the reduced representation. In that sense, the map $f : \mathbb{R}^n \to \mathbb{R}^m$ is *implicit*: it is realized by nearest-prototype assignment plus fixed grid coordinates.

This "discrete embedding" view is the reason SOMs are useful for visualization and clustering at the same time: nearby grid locations tend to represent nearby regions of the input space, but grid distance is only an *approximate* proxy for geometry. Read the map primarily as neighborhood structure and qualitative ordering, not as a metric-preserving chart.

How to read a SOM map as a feature representation

- **Out-of-sample mapping:** a new point maps to its BMU index $c(\mathbf{x})$ and grid coordinate $\mathbf{r}_{c(\mathbf{x})}$.
- **What transfers from PCA/MDS:** you can still ask whether neighborhoods are preserved and whether the map is stable across runs.
- **What does not transfer:** do not treat grid distance as true Euclidean distance in data space; the representation is discrete and only approximately geometry-preserving.

9.5 Self-Organizing Maps (SOMs): Introduction

Self-Organizing Maps (SOMs), also known as Kohonen maps, sit at a practical intersection: they behave like a prototype-based clustering method, but the prototypes are arranged on a 2D grid so you can inspect how the dataset organizes itself. Unlike supervised neural networks, SOMs learn without explicit target outputs or labels. Instead, they organize a bank of prototype vectors so that nearby units on the grid tend to represent similar inputs.

SOM at a glance

What it learns: Prototype vectors $\mathbf{w}_i$ in input space, arranged on a fixed 2D grid so that neighboring units tend to represent neighboring regions of the data (topographic mapping).
Knobs you actually tune: Map size/topology, the learning-rate schedule $\alpha(t)$, the neighborhood width $\sigma(t)$ and its decay, and the distance metric (typically squared Euclidean).
A practical starting point: A 2D rectangular grid, squared Euclidean distance, exponential decays for $\alpha(t)$ and $\sigma(t)$, and a number of units comparable to (or slightly larger than) the number of clusters you expect to resolve.
Common ways to fool yourself: Using a map that is too small, shrinking $\alpha(t)$ or $\sigma(t)$ too quickly (the map freezes before it organizes), and reading grid distance as if it were a true metric in data space.

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 Architecturally, a SOM pairs an input vector $\mathbf{x} \in \mathbb{R}^n$ with a usually two-dimensional grid of units. Each unit i has (i) a fixed grid coordinate $\mathbf{r}_i = [u_i, v_i]^\top$ with $u_i, v_i \in \mathbb{Z}$, and (ii) a prototype (codebook) vector $\mathbf{w}_i \in \mathbb{R}^n$. The coordinates $\mathbf{r}_i$ define proximity *on the grid* and enter the neighborhood function (Section 9.16); the prototypes $\mathbf{w}_i$ are what you compare to data points when you pick the winner.

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 embed-

ding.

Key Concept: Topographic Mapping The fundamental idea is simple: inputs that are similar in the original space should activate units that are close to each other on the map. In practice, that means if $\mathbf{x}_1$ and $\mathbf{x}_2$ are close in $\mathbb{R}^n$, then their best matching units should end up near each other on the 2D grid. The neighborhood update is the mechanism that encourages this: when one unit wins, its neighbors are pulled in the same direction, so "similar inputs" repeatedly shape the same local region of the map.

If you want a compact formal statement, let $\mathcal{N}_\epsilon(\mathbf{x}) = \{\mathbf{z} \mid \|\mathbf{z} - \mathbf{x}\|_2 < \epsilon\}$ be a small neighborhood in input space. SOM training aims (approximately) to map $\mathcal{N}_\epsilon(\mathbf{x})$ into a small neighborhood around the BMU on the grid (see Kohonen, 2001 for details and caveats). The guarantee is not exact; the engineering goal is a stable qualitative ordering that makes the map readable.

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.

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).

Author's note: distance is part of the model

The BMU rule is only as sensible as your distance. Euclidean distance is the default because it makes the update rule simple, but it assumes features are commensurate. In practice you almost always normalize inputs (and sometimes whiten them) so that one coordinate does not dominate the match by sheer scale. If your notion of "similar" is directional rather than magnitude-based, cosine distance is often a better choice; if covariance is strongly anisotropic, a Mahalanobis distance can be justified.

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.

This iterative process causes the map to self-organize, with neurons specializing to represent clusters or features of the input space.

9.7 Mathematical Formulation of SOM

Let the input space be $\mathcal{X} \subseteq \mathbb{R}^n$, and the output map be a grid of neurons indexed by i, each with weight vector $\mathbf{w}_i \in \mathbb{R}^n$.

Best Matching Unit (BMU) We reuse the BMU definition in (9.3); the same squared-distance criterion carries into the formal derivation here.

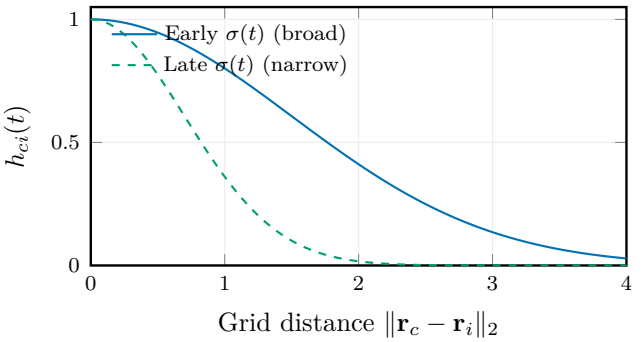

Figure 37: Gaussian neighborhood weights in SOM training (schematic). Early iterations use a broad kernel so many neighbors adapt; later iterations shrink the neighborhood width $\sigma(t)$ so only units near the BMU update.

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 grid coordinates of neuron i and $\sigma(t)$ is the neighborhood radius that decreases monotonically with t. A common schedule is an exponential decay:

$$\sigma(k) = \sigma_0 e^{-k/\tau}, \tag{9.6}$$

where k counts updates and τ sets how quickly the neighborhood shrinks. Early in training $\sigma(t)$ is large, encouraging broad cooperation; as $\sigma(t)$ shrinks, only neurons near the BMU continue to adapt (Figure 37).

9.8 Kohonen Self-Organizing Maps (SOMs): Network Architecture and Operation

So far we have described SOMs as an algorithmic loop: find the BMU, update the BMU and its neighbors, and anneal the step size and neighborhood width over time. If you prefer to think in terms of a network diagram, this subsection restates the same story as a concrete architecture you can trace and implement.

Network Structure The map is a fixed 2D grid of units. Each unit i stores a prototype vector $\mathbf{w}_i$ in input space and has a fixed grid coordinate $\mathbf{r}_i$. Every

input is compared against every stored prototype.

Mapping and Competition For a given input $\mathbf{x}$, the units compete by similarity (or distance) between $\mathbf{x}$ and $\mathbf{w}_i$. The closest unit wins; this is the BMU rule in (9.3).

Weight Update Rule Only the winning unit and its neighbors on the grid update; the update rule is (9.4). The important qualitative effect is that one input does not just move one prototype; it moves a small *patch* of prototypes in the same direction, which is what makes the grid organize into a readable map rather than a bag of unrelated cluster centers.

9.9 Example: SOM with a 3×3 Output Map and 4-Dimensional Input

To make the symbols concrete, consider a SOM whose inputs live in $\mathbb{R}^4$ and whose output map is a 3×3 grid. Each unit $i \in \{1, \ldots, 9\}$ stores a prototype vector $\mathbf{w}_i \in \mathbb{R}^4$. For a single input $\mathbf{x} = [x_1, x_2, x_3, x_4]^\top$, the job of the map is to (i) decide which prototype matches best, then (ii) move that prototype *and a small neighborhood around it* toward $\mathbf{x}$.

Feedforward Computation For a given input $\mathbf{x}$, each neuron computes a similarity score. Two common choices are:

$$y_i = \mathbf{w}_i^\top \mathbf{x} \qquad \text{(dot-product similarity)}, \tag{9.7}$$

$$d_i = \|\mathbf{x} - \mathbf{w}_i\|_2^2 \qquad \text{(squared Euclidean distance)}. \tag{9.8}$$

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 $-d_i$):

$$c = \begin{cases} \arg\max_i y_i, & \text{if similarities are measured via (9.7)}, \\ \arg\min_i d_i, & \text{if distances are used as in (9.8)}. \end{cases}$$

In practice the distance form is the most common in SOM code. If you do use dot-product similarity, normalize inputs (and often prototypes) so that "largest dot product" corresponds to your notion of "closest."

Weight Initialization and Update Weights $\mathbf{w}_i$ are typically initialized randomly (often by sampling from the data) or by a PCA-style seeding if you want a stable orientation. One training step then has a predictable shape: compute the match scores (usually d_i), pick the BMU c, compute neighborhood weights from the grid distances $\|\mathbf{r}_i - \mathbf{r}_c\|$, and apply the update in (9.4) to the BMU and a small neighborhood patch. For a fully worked numeric update with actual numbers, see the earlier *Tiny numeric step (online update)* box; the point of the present 3×3 example is to keep the dimensions explicit so you can track what gets computed where.

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 grid coordinates $\mathbf{r}_i \in \mathbb{Z}^2$ introduced for the neighborhood kernel serve as the geometry of the output map; 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.

9.10 Key Properties of Kohonen SOMs

- **Fixed output dimension:** The grid 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 grid neighbors update together.

Discriminant Function and Similarity Measures The discriminant measures in (9.7) (dot-product similarity) and (9.8) (squared Euclidean distance) are the same ones used in the earlier example; we reuse them here, favoring the distance form when deriving updates.

Weight Update Rule Once the winning neuron c is identified, WTA is the SOM update (9.4) with a collapsed neighborhood: set $h_{ci}(t) = 1$ for $i = c$ and $h_{ci}(t) = 0$ otherwise. The learning rate $\alpha(t)$ still controls step size so the winner 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 (9.4) with $h_{ci}(t)$ collapsed to a winner-only update.

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 $j = 1, 2, 3$.

Input vectors Let the four inputs be

$$\mathbf{x}_1 = \begin{bmatrix} 0.1 \\ 0.3 \\ 0.4 \\ 0.2 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} 0.8 \\ 0.7 \\ 0.2 \\ 0.1 \end{bmatrix}, \quad \mathbf{x}_3 = \begin{bmatrix} 0.2 \\ 0.2 \\ 0.1 \\ 0.9 \end{bmatrix}, \quad \mathbf{x}_4 = \begin{bmatrix} 0.4 \\ 0.6 \\ 0.5 \\ 0.3 \end{bmatrix}.$$

We walk through one update explicitly for $\mathbf{x}_1$; the remaining inputs follow the same steps.

Step 1: pick the winner for $\mathbf{x}_1$. Using squared Euclidean distance $d_j = \|\mathbf{x}_1 - \mathbf{w}_j(0)\|_2^2$,

$$d_1 = (0.2-0.1)^2 + (0.3-0.3)^2 + (0.5-0.4)^2 + (0.1-0.2)^2 = 0.03,$$
$$d_2 = \|\mathbf{x}_1 - \mathbf{w}_2(0)\|_2^2 = 0.14, \qquad d_3 = \|\mathbf{x}_1 - \mathbf{w}_3(0)\|_2^2 = 0.13.$$

So the winner is $c = \arg\min_j d_j = 1$.

Step 2: update the winner. At $t = 0$ the learning rate is $\alpha(0) = 0.3$, and only the winner updates:

$$\mathbf{w}_1(1) = \mathbf{w}_1(0) + \alpha(0)\big(\mathbf{x}_1 - \mathbf{w}_1(0)\big) = \begin{bmatrix} 0.17 \\ 0.30 \\ 0.47 \\ 0.13 \end{bmatrix}.$$

For the next input, $t = 1$ so $\alpha(1) = 0.3 \times 0.5 = 0.15$, and the same computation repeats with the new winner.

9.13 Winner-Takes-All Learning Recap

The WTA selection and update were defined in Section 9.11; the numerical example above uses the same BMU criterion (9.3) and the winner-only form of the update in (9.4).

How WTA relates to SOMs. You can view WTA learning as a limiting case of SOM training where the neighborhood collapses to a single unit: $h_{ci}(t) = \mathbf{1}[i = c]$. Adding a nontrivial neighborhood $h_{ci}(t)$ is what turns pure prototype learning into a *topographic* map: neighbors are encouraged to represent similar inputs, and the grid becomes a structured visualization rather than an unstructured codebook.

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.

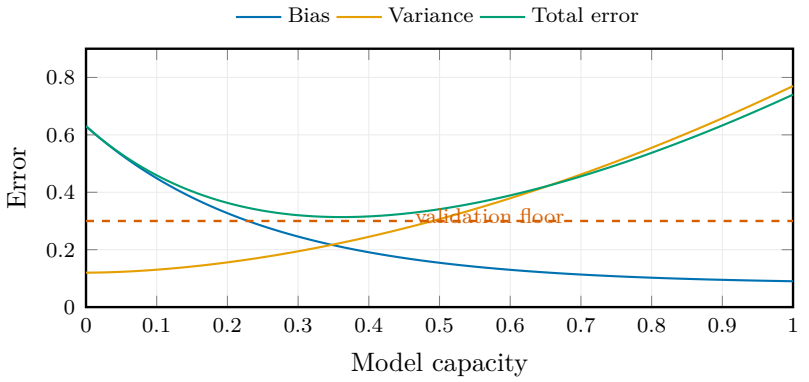

Figure 38: Illustrative bias–variance trade-off when sweeping SOM capacity (number of units or kernel width). The optimum appears near the knee where bias and variance intersect.

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 grid resolution or keeping the kernel width large for too long can overfit local noise. Figure 38 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 39. 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 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 40, helps reveal when the map is

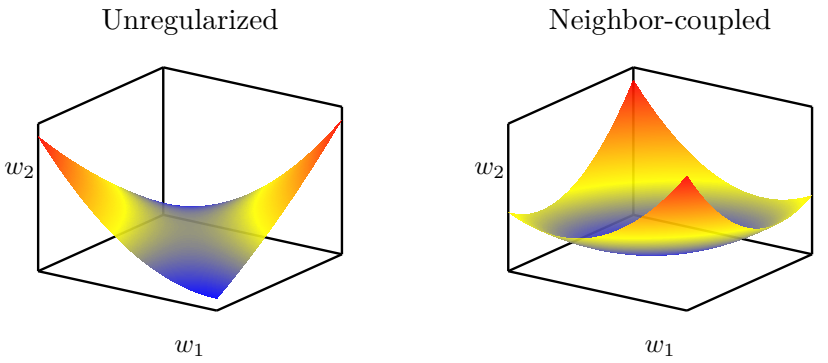

Figure 39: Regularization smooths an objective surface (schematic). Coupling neighboring prototypes (right) yields wider, flatter basins than the jagged unregularized landscape (left).

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 grid (default: 4-neighbor connectivity), capturing topology preservation. Both metrics reappear in later figures; we monitor QE for representation quality and TE for magnification distortions.

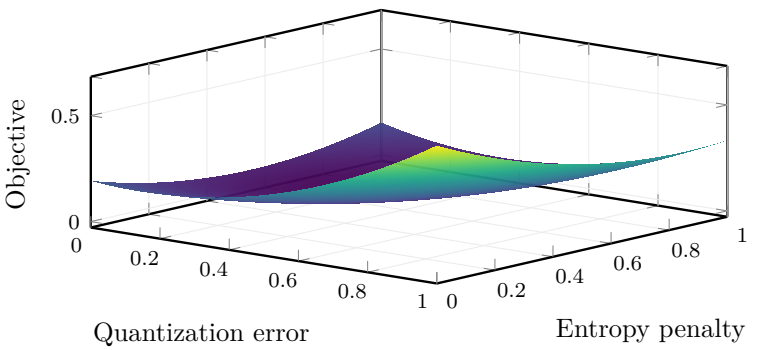

Figure 40: Illustrative objective surface combining quantization error and an entropy-style regularizer (a modern SOM variant; for example, a negative sum of $p \log p$ over unit usage). Valleys arise when prototypes cover the space evenly; ridges highlight collapse or poor topological preservation.

Tiny numeric check: QE vs. TE (they can disagree)

To see what these metrics measure, consider a 2×2 SOM grid with fixed coordinates $\mathbf{r}_1 = [0, 0]$, $\mathbf{r}_2 = [1, 0]$, $\mathbf{r}_3 = [0, 1]$, $\mathbf{r}_4 = [1, 1]$ (4-neighbor adjacency). Let the prototypes in input space be

$$\mathbf{w}_1 = [0, 0], \quad \mathbf{w}_2 = [2, 0], \quad \mathbf{w}_3 = [0, 2], \quad \mathbf{w}_4 = [0.25, 0.25].$$

Notice that $\mathbf{w}_4$ is close to $\mathbf{w}_1$ in input space, even though units 4 and 1 are diagonal neighbors on the grid.
Now evaluate four inputs, using squared Euclidean distance to pick the best and second-best matches:

$$\mathbf{x}_1 = [0.30, 0.30], \ \mathbf{x}_2 = [1.80, 0.20], \ \mathbf{x}_3 = [0.20, 1.70], \ \mathbf{x}_4 = [0.05, 0.05].$$

The BMU/second-BMU pairs are $(4, 1)$, $(2, 4)$, $(3, 4)$, $(1, 4)$. Under 4-neighbor adjacency, pairs $(4, 1)$ and $(1, 4)$ are *not* adjacent (diagonal), so $\mathrm{TE} = 2/4 = 0.5$. Meanwhile the average distance to the BMU is $\mathrm{QE} \approx 0.1962$.
The lesson is that QE is a "prototype fit" score, while TE is an "ordering" score. You can improve QE by moving prototypes toward the data while still tearing the map if nearby grid units do not represent nearby regions of the input space.

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 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.

Stopping criteria. Because stochastic updates can eventually increase topographic error, it is standard to stop training once a moving-average validation curve plateaus. Figure 41 shows the canonical trend: fast initial improvement followed by saturation.

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.

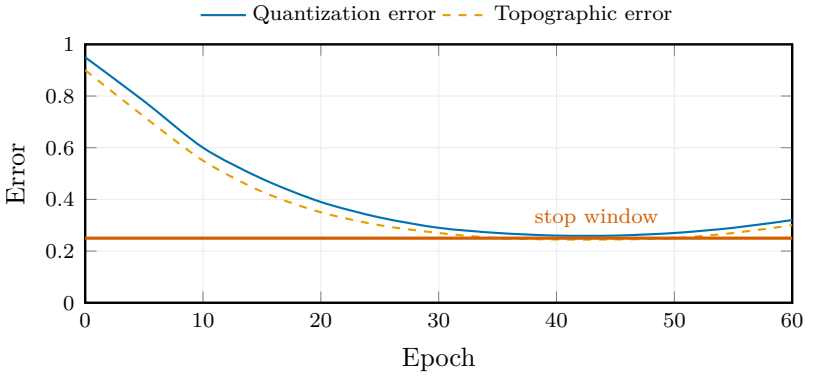

Figure 41: Illustrative validation curves used to identify an early-stopping knee. When both quantization and topographic errors flatten (shaded band), further training risks map drift.

- The resulting clusters correspond to hard assignments, so boundaries between codebook vectors are sharp with little smoothing across neighboring neurons.

There is also a very practical failure mode: you can drive the prototypes toward the data (so QE improves) while the *grid organization* never really forms. If you start with $\sigma(t) \approx 1$ from the very beginning (or you drop neighborhood updates entirely), nearby grid locations can end up representing unrelated regions of the input space. The map looks like a shuffled deck: the U-Matrix becomes speckled, and TE stays stubbornly high because the second-best match for a point often lives far away on the grid. The fix is not a new objective; it is simply the training schedule: use a broad neighborhood early so large-scale ordering can settle, then shrink $\sigma(t)$ so the map can refine without smearing away detail. More importantly for SOMs, plain WTA gives you *prototypes* but not necessarily a readable *map*. If only the winner moves, nothing forces neighboring grid units to represent neighboring regions of the data; the prototypes can end up arranged arbitrarily across the grid, and small changes in initialization can produce very different-looking maps. Cooperation is the fix: early in training you use a broad neighborhood (large $\sigma(t)$) so an input shapes a local patch rather than a single unit, which encourages global ordering; later you shrink $\sigma(t)$ so the map can refine without smearing away detail. The geometric effect of these limitations is easiest to see in Figure 42: the left panel shows the

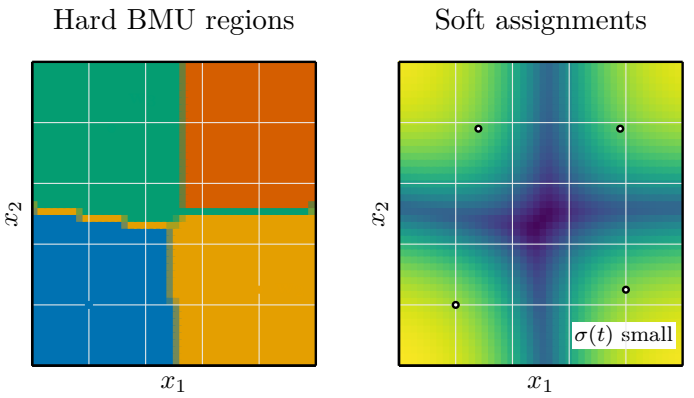

Figure 42: Voronoi-like regions induced by fixed prototypes in input space (left) and the corresponding soft assignments after sharpening the neighborhood kernel (right). Softer updates blur the decision frontiers and reduce jagged mappings between adjacent units (schematic illustration).

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 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 on the grid. 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 grid 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.

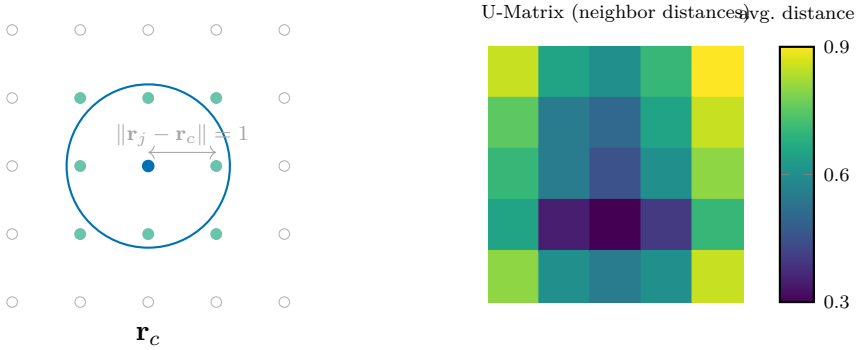

Figure 43: Left: a 5-by-5 SOM grid with the best matching unit (blue) and neighbors inside the Gaussian-kernel radius (green). Right: a toy U-Matrix (grayscale-safe colormap) showing average distances between neighboring codebook vectors; larger distances suggest likely cluster boundaries. Treat a U-Matrix as a qualitative boundary hint unless preprocessing and scaling are fixed.

Weight Update with Neighborhood Cooperation The grid structure and how the best matching unit (BMU) influences nearby neurons are visualized in Figure 43. The U-Matrix on the right provides a quick diagnostic for cluster boundaries during training.

The weight update rule generalizes to:

$$\mathbf{w}_j(t+1) = \mathbf{w}_j(t) + \alpha(t)\, h_{jc}(t)\, (\mathbf{x} - \mathbf{w}_j(t))\,, \tag{9.9}$$

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:

$$h_{jc}(t) = \begin{cases} 1, & j = c \\ \in (0,1), & j \in \mathcal{N}(c), j \neq c \\ 0, & \text{otherwise} \end{cases}$$

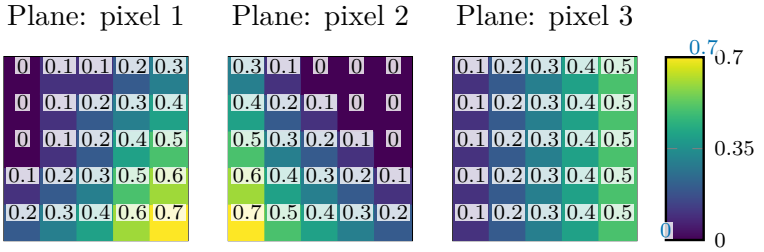

Figure 44: Component planes for three features on a trained SOM (toy data). Each plane maps one feature across the grid; aligned bright/dark regions reveal correlated features and complement the U-Matrix in Figure 43. Interpret brightness comparatively within a plane rather than as an absolute scale.

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 grid:

$$h_{jc}(t) = \exp\left(-\frac{\|\mathbf{r}_j - \mathbf{r}_c\|^2}{2\sigma^2(t)}\right), \tag{9.10}$$

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).

9.17 Example: Neighborhood Update Illustration

Suppose the output neurons are arranged in a 2D grid as shown schematically in Figure 45, 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

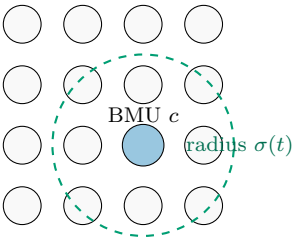

Figure 45: SOM grid with the best-matching unit (BMU) highlighted in blue and a dashed neighborhood radius indicating which prototype vectors receive cooperative updates (schematic).

σ around c.

Each neuron j in $\mathcal{N}(c)$ updates its weight vector according to (9.9), 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

At this point the SOM training story is complete: you have a similarity rule (to pick a winner), a neighborhood rule (to decide who else learns), and schedules that anneal both over time. The only subtlety is that these pieces interact. A large neighborhood early is what gives you global organization; a small neighborhood late is what lets prototypes settle into fine detail without tearing the map.

Recall the weight update rule for neuron j at time step t:

$$\Delta \mathbf{w}_j(t) = \alpha(t)\, h_{j,c}(t)\, [\mathbf{x}(t) - \mathbf{w}_j(t)]\,. \tag{9.11}$$

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:

$$h_{j,c}(t) = \exp\left(-\frac{\|\mathbf{r}_j - \mathbf{r}_c\|^2}{2\sigma^2(t)}\right). \tag{9.12}$$

where:

- $\mathbf{r}_j$ and $\mathbf{r}_c$ are the positions of neurons j and c on the SOM grid.
- $\sigma(t)$ 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:

$$\alpha(t) = \alpha_0 \exp\left(-\frac{t}{\tau_\alpha}\right), \tag{9.13}$$

$$\sigma(t) = \sigma_0 \exp\left(-\frac{t}{\tau_\sigma}\right), \tag{9.14}$$

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 One run of SOM training is repetitive in a good way: you keep presenting inputs, keep finding the BMU, and keep nudging a neighborhood patch until the map stops moving. The following pseudocode is the same loop written in a way you can implement directly.

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 grid; 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 stabilizes faster than purely online updates.

I find it useful to tell the SOM story in *three stages*, even though the code is often written as a longer checklist. The stages are: **Initialization** (seed the prototype grid), **Competition** (pick a best-matching unit for each input), and **Cooperation** (update a neighborhood and decay $\alpha(t)$ and $\sigma(t)$). The six-step procedure in Section 9.19 is simply one way to operationalize those stages when you implement the algorithm and decide when to stop.

9.20 Applications of Kohonen Self-Organizing Maps

Kohonen SOMs show up when you want *both* a prototype model and a picture you can reason about. In practice they tend to be used in three roles:

- **Clustering / regime discovery:** Group similar points without supervision and summarize each region by a prototype vector (e.g., operating modes in telemetry features).
- **Exploratory reduction:** Map high-dimensional data onto a 2D grid index for visualization and exploratory analysis. You do not get a continuous coordinate system; you get a discrete map that is often easier to inspect.
- **Visualization diagnostics:** Use U-Matrices and component planes as "instrument panels" that reveal boundaries, smoothness, and which input

features drive the organization.

Author's note: when a SOM is the wrong tool

If your only goal is to assign points to K groups and summarize each group by a center, k-means is usually the simpler baseline. If you want a continuous low-dimensional chart with a clear linear story, start with PCA; if you need a global distance-preserving embedding, MDS is a better conceptual match. If your goal is a local-neighborhood visualization and you accept distortion of global geometry, methods like UMAP/t-SNE are often more visually dramatic.

I reach for a SOM when I want three things at once: (i) prototypes I can inspect, (ii) a fixed 2D organization that is stable enough to compare across runs, and (iii) diagnostics (U-Matrix/component planes, QE/TE) that tell me whether the picture is trustworthy. If you cannot define a sensible distance (or you cannot normalize features so distance means something), a SOM will happily organize noise.

Application example: website sessions as behavioral modes

Consider a website where the system you are trying to understand is not a motor or a circuit, but a stream of user sessions. Each session becomes a data point $\mathbf{x}$: dwell time, number of pages, scroll depth, time-to-first-interaction, device class, referrer type, returning vs. new, and a few content signals (e.g., the category of the first page visited, or an embedding of the landing-page text). You may not have labels for "intent," but you do expect recurring modes: quick bounce, comparison shopping, documentation lookup, checkout flow, support troubleshooting, and so on. A SOM is useful here because it gives you prototypes $\mathbf{w}_i$ (representative session fingerprints) *and* a 2D grid you can inspect. After training, each session maps to a best-matching unit (BMU). Dense patches of mapped sessions indicate common behavior; ridges in the U-Matrix suggest boundaries where neighboring prototypes are far apart, which often corresponds to genuinely different behaviors under your feature choices. Component planes then turn the picture into an explanation: you can see, for example, that one boundary is mostly driven by dwell time and scroll depth, while another is driven by referrer and device. In practice you read these plots as heat maps over the 2D grid: different behaviors show up as different regions you can name, compare, and inspect.

Recommendation: treat preprocessing as part of the model. Normalize continuous features, log-scale heavy-tailed counts, and be deliberate about how you represent categorical variables (one-hot, grouped buckets, or a small embedding) so Euclidean distance matches your notion of similarity.

Audit hook: rerun with a few seeds and a time-based split. If the map reorganizes completely across runs or across weeks, you are seeing instability or seasonality rather than durable structure.

Application example: mapping page archetypes and performance drivers

A second, complementary use is to map *pages* (or queries) rather than sessions. Here each data point $\mathbf{x}$ describes a URL or content item: a text embedding of the page, topic/category tags, layout signals (word count, number of images), and aggregate behavior metrics (click-through rate (CTR) from search, median dwell time, bounce rate, conversion rate, and the fraction of visits from mobile). You are effectively asking: which pages are similar in content and behavior, and where do the sharp breaks live? A SOM turns this into a visual inventory. Pages with similar content and similar usage patterns compete for the same region of the grid, and the U-Matrix highlights boundaries between page families (e.g., "documentation-like" pages versus "marketing-like" pages). Component planes are the practical payoff: they show which variables line up with those boundaries. If one region lights up on "mobile fraction" and "bounce rate," you immediately get a hypothesis about responsiveness or page speed. If another region is coherent in "topic embedding" but not in conversion, you have found a content family that is consistent but not effective.
Recommendation: separate what you want to *organize by* (content similarity) from what you want to *diagnose* (performance metrics). A simple approach is to train the SOM on content-heavy features, then overlay performance as color/slices; otherwise the map can end up organizing primarily by whatever has the largest numeric range.
Audit hook: overlay slices (language, device, traffic source). If the map's dominant structure is "source domain" or "mobile vs. desktop," that may be a useful finding, but it is also a warning that your feature design is driving the story.

Relation to k-means and modern variants When the neighborhood is collapsed to a single winner (no neighbors), the update reduces to a k-means-style prototype move without any notion of grid organization (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 (theory lens): 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). One qualitative takeaway is *magnification*: dense regions of the data tend to attract more units of the map.

A practical recipe (engineering lens): If I have no prior, I treat these as starting points, not rules. Use $5\sqrt{N}$–$10\sqrt{N}$ units when unsure; hex grids reduce anisotropy; wrap-around (toroidal) grids reduce edge effects. Initialize $\mathbf{w}_j$ from data or along the first two PCs; pick $\alpha_0 \in [0.1, 0.5]$ and decay toward $\alpha_{\min} \approx 10^{-3}$; set σ_0 to roughly the map radius and decay toward 1–1.5 cells.

Related and growing variants

Neural Gas / Growing Neural Gas (Martinetz et al., 1993; Fritzke, 1994) drop the fixed grid and instead learn neighborhood structure (and, in the growing variants, add units dynamically). **Generative topographic mapping (GTM)** (Bishop et al.) provides a probabilistic, topographic embedding with likelihoods/uncertainty. I like to remember the landscape this way: k-means gives prototypes with no organization; SOMs add a fixed organizing prior; these variants let the organization itself adapt.

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 or vector-index libraries such as 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 em-

beddings, see Chapters 11 and 13; 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.

Key takeaways

- SOMs perform topology-preserving vector quantization on a discrete grid.
- A shrinking neighborhood and decaying learning rate drive coarse-to-fine organization.
- U-Matrices and quantization/topographic errors are practical diagnostics for convergence.

Minimum viable mastery.

- Given x, compute the BMU, write the weight update $w_i \leftarrow w_i + \alpha(t)h_{ci}(t)(x - w_i)$, and explain how h_{ci} enforces cooperation.
- Interpret a U-Matrix and component planes as diagnostics for neighborhood structure and convergence.
- Choose sensible $\alpha(t)$ and $\sigma(t)$ schedules and justify them using QE/TE validation curves.

Common pitfalls.

- Using a map that is too small or a neighborhood decay that is too aggressive (tearing the topology).
- Skipping input normalization, causing prototypes to track scale rather than structure.
- Treating grid distance as a true metric in data space (SOM topology is approximate, not exact geometry).
- Over-interpreting a single run without checking QE/TE plateaus and sensitivity to initialization.

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.

If you are skipping ahead. Keep the notion of an "energy landscape" from this chapter in mind: Chapter 10 makes that idea explicit with a Lyapunov energy, and later attention models (Chapter 14) can be read as learned, content-based neighborhood selection.

Where we head next. Chapter 10 extends the unsupervised thread with energy-based associative memory, complementing SOM topology learning with retrieval dynamics that foreshadow later attention-style mechanisms.

10 Hopfield Networks: Introduction and Context

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. Figure 1 marks this as the energy-based branch. The debug mindset carries over, but the diagnostics change: SOMs are easiest to audit through map geometry (U-matrix/component planes), while Hopfield recall is easiest to audit through an energy trace and overlap with the intended memory.

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.

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

Feedforward networks compute a single forward map: once an input has passed through the layers, the computation ends. That is the right abstraction for static input→output tasks, but it is not a natural model of memory: nothing inside the network persists unless you explicitly feed it back in.

Recurrent networks add that feedback. Cycles let the current state influence the next state, which is exactly what you want for sequences and recall—and also exactly what makes the dynamics harder to reason about. Hopfield's contribution is to keep the recurrence, but constrain it so the system settles instead of wandering.

Challenges with general recurrent networks The moment you add feedback, you inherit a dynamical system. That buys you memory, but it also means the network can (i) fail to settle and instead oscillate, (ii) end up in different states from tiny changes in the starting point, and (iii) become harder to train because learning signals must propagate through time (and that path can vanish or explode). Historically, these issues pushed many applications toward feedforward models unless recurrence was the cleanest abstraction for the task.

Then vs. now: energy-based memory in context

Classical Hopfield networks are the clean case: binary states, symmetric weights, and an explicit Lyapunov energy give you a convergence guarantee. The intuition that survives is the one we care about: memory is an energy landscape, and recall is descent toward an attractor (pattern completion). Modern systems borrow the geometry without necessarily inheriting the guarantee. Many "memory" mechanisms are differentiable, learned from data, and content-addressable at scale; the symmetry that makes Hopfield proofs work is not always present. Keep the attractor picture, but do not over-assume the math.

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

In 1982, John Hopfield introduced a special class of recurrent networks by putting engineering constraints on the feedback loop (Hopfield, 1982). The constraints are simple enough to implement, and each one buys you a piece of the convergence story.

First, make the weights symmetric and remove self-loops:

$$w_{ij} = w_{ji} \quad \forall i, j, \tag{10.1}$$

$$w_{ii} = 0 \quad \forall i. \tag{10.2}$$

Second, keep neuron states binary, $s_i \in \{+1, -1\}$, so the network evolves by flipping bits rather than flowing through a continuous activation curve.

Under these choices you can define a Lyapunov energy $E(\mathbf{s})$ (a scalar function that decreases under the update) and choose an asynchronous (one-neuron-at-a-time) update rule so that each flip never increases E. That is the key move: the

network becomes a descent process in a finite state space, so it must settle at a fixed point. Those fixed points are the stored patterns (and their complements), plus whatever spurious attractors appear when the network is heavily loaded.

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 $\mathbf{s} = 2\mathbf{x} - \mathbf{1}$ and $\mathbf{x} = (\mathbf{s} + \mathbf{1})/2$; 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 $\mathbf{x} = [1,0,1,0]$ in the $\{0,1\}$ encoding maps to $\mathbf{s} = 2\mathbf{x} - \mathbf{1} = [+1,-1,+1,-1]$; conversely $\mathbf{s} = [-1,+1,+1]$ corresponds to $\mathbf{x} = [0,1,1]$.

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 $\tfrac{1}{2}$ 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 $b_i = \theta_i$.

Engineering lens: energy as an objective (not a metaphor)

It helps to read $E(\mathbf{s})$ the same way you read a loss in supervised learning: a scalar objective defined over a space of states. Here the states are discrete ($\mathbf{s} \in \{-1, +1\}^N$), so the "optimization" is not gradient descent; it is a sequence of coordinate updates (flip one bit at a time) that never increase E under Hopfield's symmetry constraints.
This is why the update rule matters. The weights define the energy landscape, and the asynchronous update is a descent procedure on that landscape. Local minima are stable memories; other minima (spurious attractors) are the price of capacity.

Hopfield network states and the energy view Encoding choices and the corresponding energy forms were laid out in Section 10.4 and Section 10.5. The key continuity idea is that Hopfield dynamics are not "mysterious recurrence": with symmetric weights, the update rule becomes a descent process on $E(\cdot)$. We make this concrete in three passes: define what "stable" means, walk one tiny numeric trace, then prove the one-step claim ($\Delta E \leq 0$) that guarantees convergence under asynchronous updates.

10.6 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 using the sign rule in (10.4); for $\{0, 1\}$ encodings use the corresponding thresholded version. The pseudo-code below makes the asynchronous sweep explicit for later convergence arguments.

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 (10.10), so the loop converges to a local minimum of (10.5).

For the formal monotonicity proof and the synchronous-update caveat, see Section 10.9.

Implementation checklist (debugging a Hopfield net)

- **Weight constraints:** enforce $W = W^\top$ and $w_{ii} = 0$. If either is violated, the classical energy argument does not apply.
- **Energy definition:** implement (10.5) with the $\frac{1}{2}$ factor (or equivalently sum over $i < j$). Use one sign convention for thresholds/biases and stick to it.
- **Update scheme:** test with asynchronous single-neuron updates first. If you switch to synchronous updates, short cycles are possible (see Section 10.9).
- **Sign at zero:** choose a deterministic convention for sign(0) (e.g., keep the old state, or return +1) so runs are reproducible.
- **Diagnostics:** on a tiny network where you know the stored patterns, plot $E(\mathbf{s}(t))$ and overlap $m^{(\mu)}(\mathbf{s}(t))$. Under asynchronous updates and symmetric weights, E should never increase.

10.7 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

$$\begin{aligned} E(\mathbf{s}) &= -\frac{1}{2}\sum_{i=1}^{3}\sum_{j=1}^{3} w_{ij}s_i s_j \\ &= -\frac{1}{2}\Big[2\cdot 3\cdot (1)(1) + 2\cdot(-4)\cdot(1)(-1) + 2\cdot 2\cdot(1)(-1)\Big] = -5. \end{aligned} \tag{10.6}$$

State update attempts: One way to check whether $\mathbf{s} = (1, 1, -1)$ is stable is to try each single-neuron flip and recompute E. Flipping s_1 gives $E(-1, 1, -1) = 9$; flipping s_2 gives $E(1, -1, -1) = -3$; and flipping s_3 gives $E(1, 1, 1) = -1$. Each move raises the energy relative to -5, so no asynchronous update would accept it: the state is a local minimum.

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: 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 $(1, 1, 1)$), the next asynchronous update flips it back. In this state the local field at neuron 3 is

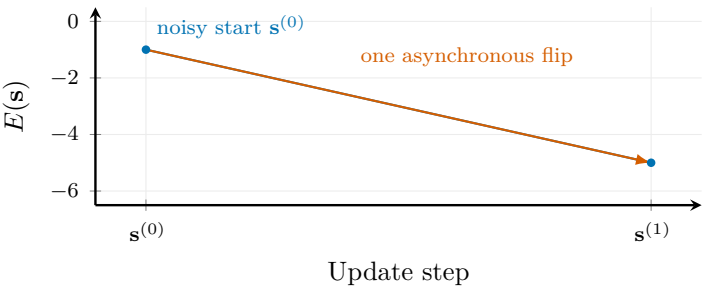

Figure 46: Hopfield energy decreases monotonically under asynchronous updates. Starting from a noisy probe $\mathbf{s}^{(0)}$, single-neuron flips move downhill until a stable memory is recovered.

$h_3 = -4(1)+2(1) = -2$, so the update rule sets $s_3 \leftarrow -1$, returning to $(1, 1, -1)$. The energy drops from -1 to -5, which is the basic content-addressable recall behavior.

As Figure 46 shows, the energy trace never increases when single neurons flip asynchronously, so the descent depicted here is exactly what the formal convergence proof below captures.

10.8 Energy Function and Convergence of Hopfield Networks

The figure and pseudo-code above already suggest the main claim: under the symmetry constraints that make $E(\cdot)$ well defined, asynchronous updates are guaranteed to move "downhill" (or stay flat) in energy. This is the formal reason Hopfield networks converge to a fixed point: the state space is finite, and you cannot decrease a bounded quantity forever.

In this subsection we prove the monotonicity step that the rest of the convergence story rests on. The only assumptions you should keep in view are the ones you would implement anyway: symmetric weights with zero diagonal, and an asynchronous update rule (one neuron at a time). When these assumptions are violated (e.g., synchronous updates or asymmetric weights), the energy argument no longer guarantees convergence.

We use the energy in (10.5); the goal is to show that each single-neuron update never increases it.

Goal: Show that asynchronous updates of neuron states always decrease (or leave unchanged) the energy E, guaranteeing convergence to a local minimum.

10.8.1 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

$$\Delta E = E_{\text{new}} - E_{\text{old}}. \tag{10.7}$$

Using (10.5), write out the energies explicitly:

$$E_{\text{old}} = -\frac{1}{2}\sum_{k=1}^{N}\sum_{l=1}^{N} w_{kl} s_k^{\text{old}} s_l^{\text{old}} + \sum_{k=1}^{N} \theta_k s_k^{\text{old}}, \tag{10.8}$$

$$E_{\text{new}} = -\frac{1}{2}\sum_{k=1}^{N}\sum_{l=1}^{N} w_{kl} s_k^{\text{new}} s_l^{\text{new}} + \sum_{k=1}^{N} \theta_k s_k^{\text{new}}. \tag{10.9}$$

Since only s_i changes, and weights are symmetric with zero diagonal $w_{ii} = 0$, the difference simplifies to

$$\begin{aligned}
\Delta E &= E_{\text{new}} - E_{\text{old}} \\
&= -\frac{1}{2}\sum_{j=1}^{N} (w_{ij} s_i^{\text{new}} s_j + w_{ji} s_j s_i^{\text{new}}) + \theta_i s_i^{\text{new}} \\
&\quad + \frac{1}{2}\sum_{j=1}^{N} \left(w_{ij} s_i^{\text{old}} s_j + w_{ji} s_j s_i^{\text{old}} \right) - \theta_i s_i^{\text{old}} \\
&= -\sum_{j=1}^{N} w_{ij} s_j \left(s_i^{\text{new}} - s_i^{\text{old}} \right) + \theta_i \left(s_i^{\text{new}} - s_i^{\text{old}} \right) \\
&= -\left(s_i^{\text{new}} - s_i^{\text{old}} \right) \left(\sum_{j=1}^{N} w_{ij} s_j - \theta_i \right).
\end{aligned} \tag{10.10}$$

Define the *local field* h_i at neuron i as

$$h_i = \sum_{j=1}^{N} w_{ij} s_j - \theta_i. \tag{10.11}$$

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 (10.10).

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. The useful connection for us is the geometry: content-addressable lookup as motion in an energy landscape. Do not confuse that bridge with the classical guarantee above: the Lyapunov proof in this chapter relies on binary states, symmetric weights, and asynchronous flips.

Continuous Hopfield networks (bridge). Modern extensions replace the binary sign activation with a smooth, often softmax-like update that keeps neuron states continuous. Storing P real-valued patterns $\{\mathbf{m}^\mu\}$, a common update takes the form

$$\mathbf{s}^{(t+1)} = \text{softmax}\Big(\beta M^\top \mathbf{s}^{(t)}\Big)\, M,$$

where M stacks the memory vectors and β controls sharpness. Read this as "soft" associative recall: the update produces a weighted combination of stored patterns, and that picture lines up closely with attention in Chapter 14. Unlike the classical binary case, you should not assume strict monotone descent unless the model is built to have a Lyapunov function.

Combining the sign update in (10.4) with (10.10) yields $\Delta E \leq 0$ for each asynchronous flip, so we omit the redundant case split here.

10.9 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? With symmetric weights you can still write the same energy $E(\mathbf{s})$, but the classic monotonicity proof is for *single-neuron* (asynchronous) updates. If you update all neurons simultaneously, two-cycles can appear.

For a concrete example, take two neurons with $\theta_1 = \theta_2 = 0$ and $w_{12} = w_{21} = 1$. Start from $(s_1, s_2) = (1, -1)$. A synchronous update computes both new states from the old pair:

$$s_1 \leftarrow \operatorname{sign}(w_{12}s_2) = \operatorname{sign}(-1) = -1, \qquad s_2 \leftarrow \operatorname{sign}(w_{21}s_1) = \operatorname{sign}(1) = 1,$$

so the state becomes $(-1, 1)$. Repeating the synchronous update maps $(-1, 1)$ back to $(1, -1)$. The network therefore oscillates in a 2-cycle, even though the energy

$$E(s_1, s_2) = -\frac{1}{2}\sum_{i,j} w_{ij}s_i s_j = -w_{12}s_1 s_2$$

stays constant across the two states.

To ensure the classical convergence guarantee, use *asynchronous* updates (one neuron at a time) under symmetric weights with zero diagonal. Synchronous updates can still work in practice, but the energy argument no longer rules out short cycles.

10.10 Storage Capacity of Hopfield Networks

A key question is: *How many memories can a Hopfield network reliably store and recall?*

$$p \approx 0.138\, n,$$

is the classical rule-of-thumb for the number of random patterns that can be stored with low error in a network of n neurons (McEliece et al., 1987). The important point is not the constant itself; it is that the capacity is only a small fraction of n. Both ξ^μ and its complement $-\xi^\mu$ are fixed points, and odd mixtures of stored patterns become spurious states as p/n grows. This low capacity is why Hopfield networks are not used as practical storage devices despite their elegant associative-memory behavior.

Stochastic updates (bridge to Boltzmann machines). A stochastic variant replaces the hard sign in (10.4) with probabilistic flips (e.g., Gibbs sampling). With symmetric weights this defines a Boltzmann distribution whose energy matches (10.5), linking Hopfield recall to the Boltzmann/energy-based models that underlie modern probabilistic neural networks.

10.11 Improving Storage Capacity via Weight Updates

Is it possible to improve storage by choosing the weights to "bake in" the memories directly? That is exactly what the classical Hebbian construction does: use stored patterns to set W, then let the state updates perform recall.

Hebbian learning rule: Given p stored patterns $\{\xi^1, \xi^2, \ldots, \xi^p\}$, each $\xi^\mu = (\xi_1^\mu, \xi_2^\mu, \ldots, \xi_n^\mu)$ with $\xi_i^\mu \in \{+1, -1\}$, the weights are set by:

$$w_{ij} = \frac{1}{n} \sum_{\mu=1}^{p} \xi_i^\mu \xi_j^\mu, \quad w_{ii} = 0. \tag{10.12}$$

This is the classical Hebbian learning rule for Hopfield networks.

What this formula is doing: Setting $w_{ii} = 0$ removes self-feedback. The factor $1/n$ keeps weight magnitudes $O(1)$ as the network grows. The sum itself is an outer-product accumulation: it encodes pairwise correlations across the stored patterns.

10.12 Example: Weight Calculation for a Single Pattern

Consider a fundamental memory pattern:

$$\xi = (1, 1, 1, -1),$$

with no thresholds ($\theta_i = 0$).

Step 1: Compute outer product Form the matrix $\mathbf{B} = \xi\xi^\top$. Each entry $B_{ij} = \xi_i\xi_j$ captures the pairwise correlation between neurons i and j.

$$\mathbf{B} = \xi\xi^\top = \begin{bmatrix} 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & -1 \\ -1 & -1 & -1 & 1 \end{bmatrix}.$$

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. Including the $1/n$ normalization from (10.12) (here $n = 4$) gives

$$\mathbf{W} = \frac{1}{4}\Big(\mathbf{B} - \mathrm{diag}(\mathbf{B})\Big) = \frac{1}{4}\begin{bmatrix} 0 & 1 & 1 & -1 \\ 1 & 0 & 1 & -1 \\ 1 & 1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}.$$

Sanity check: the stored pattern is a fixed point With zero thresholds, the local field is $\mathbf{h} = \mathbf{W}\xi$. For this one-pattern example,

$$\mathbf{h} = \begin{bmatrix} 3/4 \\ 3/4 \\ 3/4 \\ -3/4 \end{bmatrix},$$

so $\mathrm{sign}(\mathbf{h}) = \xi$. That is the basic content-addressable "snap-back": if you start near ξ, asynchronous updates push you toward it.

Numeric recall trace (two asynchronous flips)

Start from a probe with two wrong bits, $\mathbf{s}^{(0)} = (1, -1, -1, -1)$. With $\theta_i = 0$, the local field is $h_i = \sum_j w_{ij} s_j$, and the update rule (10.4) sets $s_i \leftarrow \text{sign}(h_i)$. One possible asynchronous update order (neuron 2, then neuron 3) gives:

Step	Update	Local field	New state
0	–	–	$(1, -1, -1, -1)$
1	$i = 2$	$h_2 = 0.25$	$(1, 1, -1, -1)$
2	$i = 3$	$h_3 = 0.75$	$(1, 1, 1, -1) = \xi$

The energy drops along the way: $E = 0.5 \to 0 \to -1.5$.

10.13 Finalizing the Hopfield Network Derivation and Discussion

We have already defined the Hebbian weights in (10.12), the update rule in (10.4), and the energy in (10.5). Here we focus on what those ingredients imply for retrieval basins and limitations in practice.

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:

$$\mathbf{s}(\infty) \in \{\xi^\mu, -\xi^\mu\}.$$

The same story can be read visually as motion toward a basin of attraction in an energy landscape (Figure 47).

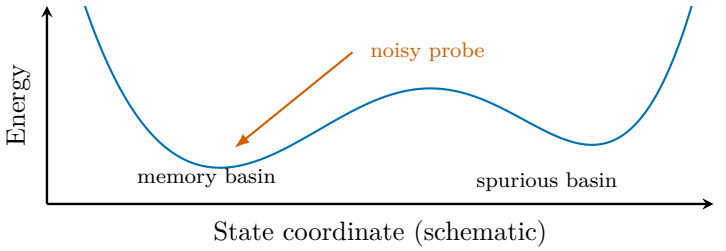

Figure 47: Energy-landscape intuition (schematic). Hopfield recall is a descent process toward a nearby basin (a stored memory), but other minima can exist and act as spurious attractors when the network is heavily loaded.

Terminology and diagnostics

Attractor. A stable fixed point of the update rule: once you reach it, further asynchronous updates do not change the state.

Basin of attraction. The set of initial states that converge to a given attractor under the chosen update scheme.

Spurious state. An attractor that is not one of the intended stored patterns (nor its complement). Spurious states are real local minima of $E(\cdot)$ created by interference among memories.

Overlap (a simple health check). For a stored pattern ξ^μ, define

$$m^{(\mu)}(\mathbf{s}) = \frac{1}{N}(\xi^\mu)^\top \mathbf{s}.$$

It ranges from $+1$ (exact match) to -1 (exact complement). Plot $E(\mathbf{s}(t))$ and $m^{(\mu)}(\mathbf{s}(t))$ versus update step to see whether recall is behaving or drifting into a spurious basin.

Limitations (what breaks first) Hopfield networks are a beautiful model to reason about, but several limitations show up quickly in practice:

- **Capacity:** with the classical Hebbian construction, reliable storage for random patterns scales like $P_{\max} \approx 0.138\, N$ for large N. Past that loading level, retrieval errors and spurious minima become common.
- **Spurious attractors:** even below capacity, the energy landscape can contain minima that are not stored patterns (or their complements). A

noisy probe can "snap" to one of these unintended fixed points.

- **Classification mismatch:** for discriminative tasks (say, digit recognition), the "nearest minimum" behavior is not the same thing as a calibrated class decision. A corrupted input can converge to the wrong stored pattern, and low energy does not imply correct label.
- **When not to use:** heavy loading (P/N large) produces a glassy landscape; scaling is $O(N^2)$ in connectivity; and the low-capacity regime makes classical Hopfield networks a poor fit for high-dimensional supervised problems compared with the ERM models in Chapter 4 or the deep models in Chapter 14.

Applications lens: memory retrieval and optimization

In the classical framing, Hopfield networks are about associative memory: you store patterns, then recover a full pattern from a partial or noisy cue (pattern completion). That is the same idea behind many information-retrieval and recognition stories: the input is a corrupted probe, and the system "snaps" to a nearby clean prototype.
There is also an optimization lens. Because $E(\mathbf{s})$ is a scalar objective over binary states, you can sometimes encode a cost function as an energy and run asynchronous updates as a heuristic descent method. This is one way Hopfield-style dynamics have been discussed for combinatorial optimization tasks (for example, variants of traveling-salesman-style objectives). The warning is the same one we give throughout this book: the algorithm will happily converge to a local minimum. Your real work is in deciding whether the energy you wrote down matches the engineering goal and whether the resulting minima are meaningful.

Example: Memory recovery (one flip) Store one pattern

$$\xi = (-1, -1, 1, -1)^T.$$

With $n = 4$, the Hebbian weights from (10.12) are

$$\mathbf{W} = \frac{1}{4}\xi\xi^\top, \qquad w_{ii} = 0,$$

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 $w_{13} = w_{31} = -0.25$). More generally, every off-diagonal weight is the scaled product of the corresponding pattern entries.

Now start from a one-bit-corrupted probe $\mathbf{s}^{(0)} = [-1, -1, 1, 1]^T$, with $\theta_i = 0$. If we update neuron 4,

$$h_4 = \sum_j w_{4j} s_j = -0.75 \quad \Rightarrow \quad s_4 \leftarrow -1,$$

so in one asynchronous flip we recover $\mathbf{s}^{(1)} = \xi$. The energy drops from $E = 0$ to $E = -1.5$ under (10.5). 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 I like Hopfield networks because they show a recurring engineering move: constrain a system until you can prove something useful, then use that proof as a debugging lens. Symmetry + no self-loops gives you an energy you can compute; asynchronous updates give you a monotone progress measure.

You will almost never deploy a classical Hopfield net as-is (capacity and

spurious minima show up fast). But the idea "memory as an objective landscape" survives, and it is the right mental model to carry into later energy-based and content-addressable mechanisms.

Connections to other chapters. Hopfield networks sit next to SOMs in spirit: both are unsupervised, but the diagnostic tools differ. SOMs give you map geometry (U-matrix/component planes); Hopfield gives you overlap and an energy trace to check whether recall is behaving. Later, attention revisits content-addressable lookup in a differentiable form.

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.

Minimum viable mastery.

- Write the energy $E(s)$, state the symmetry condition $W = W^\top$ with zero diagonal, and explain why asynchronous updates decrease E.
- Distinguish true memories from spurious attractors, and connect failure modes to loading P/N and correlation among stored patterns.
- Use overlap and energy traces as diagnostics when demonstrating recall under corruption.

Common pitfalls.

- Using asymmetric weights or nonzero self-connections (breaks the standard energy argument).
- Overloading the network and expecting clean recall (spurious minima dominate).
- Reporting a single cherry-picked recall trajectory instead of multiple corruption levels and multiple stored sets.

Exercises and lab ideas

- Implement asynchronous recall for bipolar states with symmetric W and $w_{ii} = 0$. Plot energy vs. update step and verify it never increases.
- Store P random patterns with Hebbian weights. Use a fixed random seed and a fixed corruption protocol: pick one pattern ξ^{μ}, flip k bits uniformly at random, then run asynchronous updates for a fixed budget (or until no flips occur). Sweep P/N and corruption level k; plot both energy $E(\mathbf{s}(t))$ and overlap $m^{(\mu)}(\mathbf{s}(t))$ versus step, and report when spurious attractors become common.
- Construct (or search for) a small example where synchronous updates enter a 2-cycle; compare with asynchronous updates from the same initial state.

If you are skipping ahead. The key transferable idea is the combination of a state update rule with a scalar quantity that tracks progress (energy, loss, or validation curves). That mindset carries into deep training in Chapters 6 to 11.

Where we head next. Chapter 11 turns from associative memory to deep feedforward perception, where convolution and pooling form hierarchical features. The optimization workflow is unchanged: the ERM toolkit from Chapter 3 and the training mechanics from Chapters 6 to 7 remain central. The analogy to keep in mind is the diagnostic habit: in Hopfield networks we watched an energy trace and overlap; in deep CNNs you will watch training/validation curves and slice errors. Recurrence and attention return in Chapter 12 and Chapter 14.

11 Convolutional Neural Networks and Deep Training Tools

Chapter 10 used energy to make recurrence stable: dynamics settle, memories persist, and failure has a concrete signature. Vision problems have a different flavor. The system must form a representation that is rich enough to support decisions, yet structured enough to learn from finite data.

Convolutional networks are the simplest version of that bargain. They keep the empirical risk minimization (ERM) workflow from Chapters 3 to 4 and the gradient-training machinery from Chapter 6, but make images tractable by enforcing locality and weight sharing, so the model spends capacity on patterns that recur across the grid.

There is also a useful continuity lens from earlier chapters: a convolutional filter is a learned *local template* that is reused everywhere. This is the same high-level idea as prototypes/bases in Chapters 8 and 9—but with strict weight sharing and end-to-end gradient training, so the templates are shaped by the task loss rather than chosen or tuned in isolation.

The second thread in this chapter is pragmatic. Deep models are only useful when they train. After the CNN mechanics, we transition into the deep-training toolkit (initialization, activations, normalization, regularization, and optimizers) that will reappear in Chapters 12 to 14.

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 the role of downsampling.
- Choose losses and metrics deliberately (e.g., cross-entropy vs. mean squared error (MSE); accuracy vs. calibration) for CNN-based models.
- Read a CNN as a learned feature map and connect that viewpoint to linear margins (hinge/soft-margin) and earlier kernel-style ideas in Chapter 3.
- Describe practical stabilizers (batch normalization (BN), dropout, weight decay, optimizers) and the failure signatures they address.

Design motif

Keep the same statistical learning loop from Chapters 3 to 4, but move a large fraction of the "bias" into the architecture: locality and weight sharing.

11.1 Motivation and map

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.

This is also where the training loop became practical at scale: large datasets and modern accelerators made iterative optimization feasible, and architectural priors such as locality and weight sharing made that optimization sample-efficient on image grids.

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.

How to read this chapter.

- **Core thread (CNNs):** locality + weight sharing → convolution/pooling mechanics → channels/feature maps → end-to-end classifiers.
- **Going deeper:** depth as receptive-field growth, and why stacked small kernels can see large spatial extent.
- **Training toolkit:** the stabilizers you will reuse later (initialization, activations, normalization, regularization, optimizers).

11.2 Why fully connected layers break on images

Dense feedforward networks can be competitive on small tabular problems, but on images they become data-hungry fast for two reasons: flattening throws away the grid structure, and dense connectivity explodes parameter count. The issue

is not that the model class is "wrong"; it is that the sample complexity and compute burden are badly mismatched to the structure of vision data.

An image is naturally a 2D array (or a 3D tensor with channels). For example, a single-channel (grayscale) image of size 256×256 pixels can be represented as a matrix:

$$X \in \mathbb{R}^{256\times256}.$$

To input this into a traditional feedforward network, the image must be *flattened* into a vector:

$$\mathbf{x} = \mathrm{vec}(X) \in \mathbb{R}^{65,536},$$

where $65,536 = 256 \times 256$ is the total number of pixels.

That step is not "wrong," but it is expensive. Flattening 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.

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.

The number of weights between the input and hidden layer is:

$$65,536 \times 100 = 6,553,600, \tag{11.1}$$

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 over 6 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 (a sanity check). When a model has millions of free parameters and no strong architectural prior, you are asking the data to do most of the work: it must fit the mapping and also implicitly teach the model what kinds of patterns are worth representing. A crude sanity check sometimes used in practice is to compare the number of labeled examples to the number of parameters:

$$N_{\text{samples}} \geq 10 \times N_{\text{parameters}}. \tag{11.3}$$

Read this as a warning sign, not a requirement. Inductive bias (convolutions), regularization, and augmentation reduce the effective degrees of freedom, and pretraining changes the problem entirely because you are no longer learning all features from scratch.

For the dense first layer in this example calculation, the back-of-the-envelope lands in the tens of millions of labeled images:

$$N_{\text{samples}} \gtrsim 10 \times 6{,}553{,}600 \approx 65{,}536{,}000, \tag{11.4}$$

The point is not the exact number; it is the scale mismatch between dense connectivity and the datasets most people can realistically label. Without additional structure, many very different parameter settings can fit the same training set; the architecture and the training recipe decide whether any of those settings generalize in a way you can trust.

Computational and storage constraints. Even if you could collect that many labeled examples, training cost becomes part of the model design. Each epoch is a full sweep over the dataset; each sweep costs time, memory bandwidth, and usually specialized hardware. Large models also cost money to store and to run: inference latency, memory footprint, and energy use become constraints that matter as much as accuracy. This is another reason architectural bias matters: it reduces degrees of freedom and makes it possible to spend computation on structure that transfers across images.

Overfitting risk (what to watch for). With high capacity, "training accuracy goes up" is not evidence of learning the right thing. Watch the gap between training and validation curves, check calibration (are probabilities meaningful?), and run slice audits (lighting, viewpoint, background, demographic proxies) to

detect memorization or shortcut learning early.

11.3 Sparse connectivity and parameter sharing

CNNs replace dense connections with *local receptive fields*. A unit's receptive field is simply the portion of the input that can influence that unit's value through the connections that reach it. The term is borrowed from sensory systems: a neuron "receives" evidence from a region, not from everywhere at once.

Each output unit therefore 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.

Equivariance means: if you shift the input, the feature map shifts in the same way. By contrast, invariance means the final decision changes little (or not at all) under small shifts; pooling and downsampling are common ways to trade some equivariance for more invariance at the classifier.

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.

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.

Notation handoff. Here $\sigma(\cdot)$ is used only as an activation nonlinearity, while earlier statistical chapters use σ^2 for variance. Keep Appendix D nearby when moving between probability and deep-learning sections.

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

$$(X * K)_{ij} = \sum_{u=0}^{k-1} \sum_{v=0}^{k-1} K_{uv}\, X_{i+u,\, j+v}.$$

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 accounting, and the inductive bias; the terminology is imperfect but the idea is powerful.

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.

Worked example: 1D cross-correlation values (stride + padding)

Let the input be $\mathbf{x} = [1, 2, 0, 3, 1]$, the (unflipped) filter be $\mathbf{w} = [1, 0, -1]$, padding $p = 1$, and stride $s = 2$. The padded input is $[0, 1, 2, 0, 3, 1, 0]$ and the output length is

$$L = \left\lfloor \frac{n + 2p - k}{s} \right\rfloor + 1 = \left\lfloor \frac{5 + 2 - 3}{2} \right\rfloor + 1 = 3.$$

Sliding windows of length $k = 3$ with stride 2 yield

$$\begin{aligned} y_0 &= \langle [0, 1, 2], [1, 0, -1] \rangle = -2, \\ y_1 &= \langle [2, 0, 3], [1, 0, -1] \rangle = -1, \\ y_2 &= \langle [3, 1, 0], [1, 0, -1] \rangle = 3, \end{aligned}$$

so $\mathbf{y} = [-2, -1, 3]$.

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).

11.5 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.

11.6 Channels and feature maps

Real inputs are multi-channel. A red-green-blue (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 accounting 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.

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.

Losses, metrics, and the "CNN as features" viewpoint

A useful way to read a CNN is as a learned feature map $\phi_\theta(\cdot)$ followed by a simple readout. The last layer is often just a linear classifier on top of the learned features, trained end-to-end by backpropagation.

Loss choice. For classification, the default is softmax cross-entropy because it turns scores into probabilities that can be audited (and sometimes calibrated) when the model is well-regularized. For regression-style outputs (counts, keypoints, continuous targets), squared error and its robust variants are common.

Margins and hinge. If you replace cross-entropy with a hinge-style objective, you are pushing for a margin in the learned feature space. This is the same geometric idea as the soft-margin story in Chapter 3; the difference is that kernels fix the feature map ahead of time, while CNNs learn ϕ_θ from data.

Metric choice. Accuracy is not enough when scores are used as probabilities. Track calibration (e.g., reliability curves and expected calibration error (ECE)) and slice performance, as emphasized in Chapter 4 and revisited in the audit boxes later in this chapter.

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.

At scale, this architectural prior became decisive once large labeled datasets and graphics processing units (GPUs) and training pipelines matured: the sample-efficiency and compute-efficiency gap versus classical kernel pipelines narrowed and then flipped in favor of deep convolutional stacks.

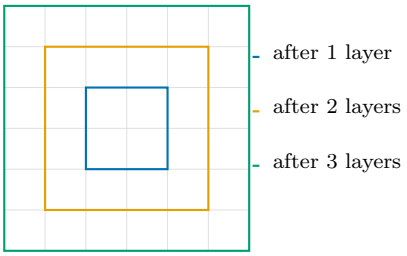

Figure 48: Receptive field growth across depth. Even with small 3×3 kernels, stacking layers expands the spatial context seen by deeper units, which is why depth can replace very large kernels.

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. Figure 48 sketches the nested growth across depth.

Quick check: receptive field growth for stacked 3×3

If each convolution uses stride 1 and a 3×3 kernel, then every new layer expands the receptive field by 2 pixels in each spatial direction. So the effective receptive field sizes go 3×3, then 5×5, then 7×7, and so on. Depth is therefore a controlled way to "see farther" without jumping to very large kernels.

With the architectural motivation in place, we now focus on how these models are trained and why deep optimization remains delicate.

11.7 Training Neural Networks: Gradient-Based Optimization

We already derived the gradient-descent training loop in Chapter 6: define a loss $\mathcal{L}$, compute gradients, then update parameters. CNNs keep the same loop. The key difference is weight sharing: a convolutional filter is applied at many spatial locations, so a single filter coefficient collects learning signal from every location where it was used.

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 $\partial \mathcal{L}/\partial w$ is computed by backpropagation: cache the forward intermediates, then apply the chain rule backward through the computation graph. In practice frameworks automate this, but the manual story is what lets you sanity-check shapes, signs, and scaling when training behaves strangely.

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.

11.8 Vanishing and Exploding Gradients in Deep Networks

Backpropagation multiplies Jacobians through many layers, so gradients can either vanish (approach zero) or explode (grow exponentially large), leading to significant training difficulties.

Mathematical intuition Consider a deep network with L layers, and focus on the error signals that get propagated backward. Let $\boldsymbol{\delta}^{(\ell)} = \partial \mathcal{L}/\partial \mathbf{z}^{(\ell)}$ denote the derivative of the loss with respect to the pre-activation at layer ℓ. A single backprop step has the form

$$\boldsymbol{\delta}^{(\ell-1)} \;=\; \Big(\mathbf{W}^{(\ell)}\Big)^{\top} \boldsymbol{\delta}^{(\ell)} \;\odot\; f'\Big(\mathbf{z}^{(\ell-1)}\Big), \tag{11.6}$$

so the early-layer signals are built from repeated multiplication by weight matrices and activation derivatives.

If the typical gain of these factors is greater than one, the signal grows as it moves backward and gradients can explode; if it is less than one, the signal shrinks and gradients can vanish. This is why initialization scale, activation choice, and normalization matter so much in deep stacks.

Consequences

- **Exploding gradients:** gradients become extremely large, causing numerical instability and making parameters diverge.

- **Vanishing gradients:** gradients become extremely small in early layers, so those weights learn slowly and the model struggles to build useful low-level 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.

Training failure signatures (symptom → likely fixes)

- **Loss becomes NaN (not a number) / diverges quickly:** lower learning rate; check data normalization; add gradient clipping; verify BN statistics (or switch norm).
- **Training loss decreases but validation stalls:** audit split hygiene (Chapter 3); add regularization (weight decay, dropout); strengthen augmentation; check shortcut features via slice tests.
- **Both training and validation improve slowly:** revisit initialization and normalization; try momentum/AdamW; confirm the objective matches the metric you care about.
- **High accuracy but poor probability decisions:** audit calibration (reliability curves, ECE) and re-threshold on held-out data rather than trusting raw scores (Chapter 4).

11.9 Strategies to Mitigate Vanishing and Exploding Gradients

Weight initialization Initialization is not just a detail: it sets the scale of activations and gradients at the start of training. A practical rule is to pick the weight variance to be inversely proportional to fan-in (and sometimes fan-out), so signals neither blow up nor fade as they pass through many layers. Xavier and

He initializations are two standard choices that implement this idea for common nonlinearities.

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 stabilizes the scale of activations during training by normalizing them (per channel) and then learning a scale and shift. The practical payoff is that fewer layers sit in saturated regimes, so gradients behave more predictably early in training. Remember that BN behaves differently at train time (mini-batch statistics) versus evaluation time (running statistics).

Gradient clipping For exploding gradients, gradient clipping limits the maximum gradient norm during backpropagation, preventing a single bad batch from producing a destructive update. It is not a substitute for good initialization and normalization, but it is a practical safety valve when training is otherwise stable.

Taken together, these tools stabilize optimization; the figures below highlight dropout, normalization, and optimizer behavior in practice.

Risk & audit

- **Train/val mismatch:** augmentation, preprocessing, and normalization must match 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.
- **Run-to-run variance:** report seeds, spread, and stopping policy; use Appendix E as the default reporting format.

As a concrete diagnostic, Figure 49 shows how dropout can flatten the validation curve relative to the no-dropout baseline.

Batch normalization. BN accelerates convergence by keeping per-channel activations on a stable scale and then letting the network relearn the best scale/shift. Figure 50 contrasts the pre- and post-normalization activation distributions; the practical point is that gradients stay in a usable range more often. In later chapters, you will see the same stability instinct show up again in LayerNorm placement choices in very deep stacks.

Adaptive optimizers. While plain SGD remains a workhorse, Adam and related methods adapt learning rates per-parameter (Kingma and Ba, 2015). Figure 51 summarizes the typical loss trajectories; Adam converges faster initially, whereas SGD+momentum often attains a slightly lower asymptote after fine-tuning.

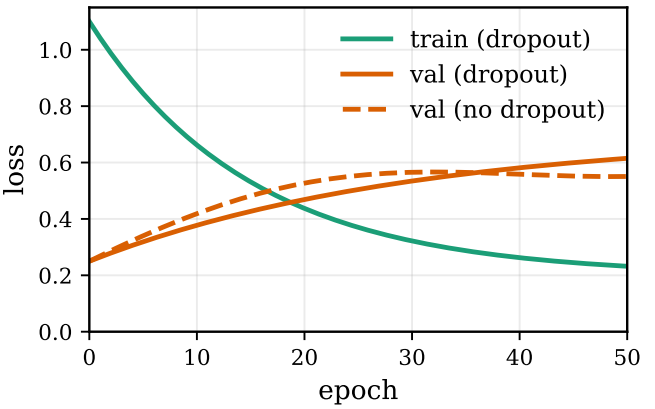

Figure 49: Illustrative dropout effect on training/validation curves. Compared to a no-dropout baseline, validation curves flatten and generalization improves; the point is the shape of the curves, not the exact values.

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.

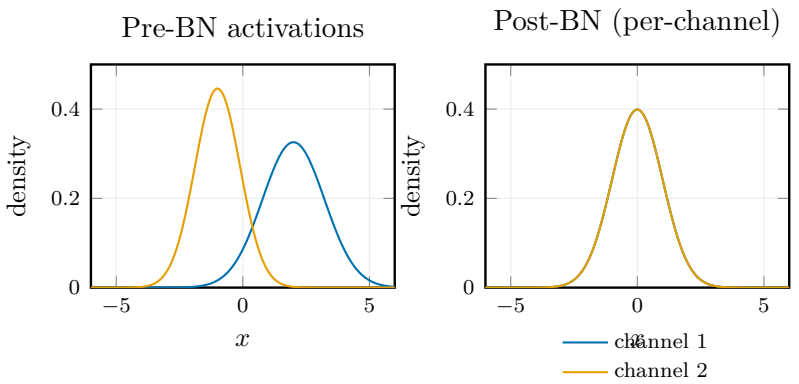

Figure 50: Batch normalization normalizes per-channel activations and then applies a learned scale and shift, which stabilizes training by keeping activation scales in a reasonable range.

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.

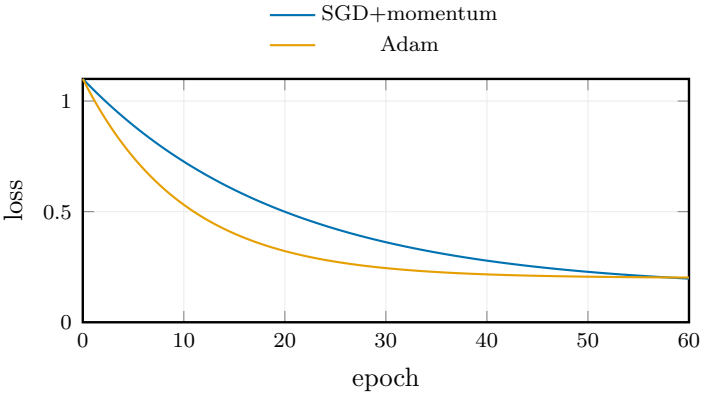

Figure 51: Representative training curves for SGD with momentum versus Adam on the same CNN. The point is the typical shape: Adam often drops loss quickly early, while SGD+momentum can be steadier later.

Derivation closure: implement, cache, and fail fast

- **Implement blocks with shapes:** treat each block (conv/BN/activation/pool) as a typed transform and keep input-output shapes explicit.
- **Cache what the backward pass needs:** pre-activations, normalization statistics, and masks should be stored, not re-derived ad hoc.
- **Fail fast:** validate shape arithmetic, then run a one-batch overfit test before full training to catch broken gradients early.
- **Report like an engineer:** publish seed, augmentation recipe, and calibration metrics with every benchmark, following Appendix E.

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.

A good checkpoint.

- Given (n, f, s, p), compute output shapes and parameter counts for a convolutional block.
- Explain equivariance vs. invariance and how stride/pooling/padding affect each.
- Describe what batch normalization and dropout do during training and how they change behavior at evaluation time.

Common pitfalls.

- Letting shape accounting drift (mismatched padding/stride) until the first dense layer fails.
- Forgetting train/eval mode for batchnorm/dropout, producing misleading validation curves.
- Treating pooling as mandatory or always beneficial; in some tasks, stride-2 convolutions or anti-aliasing are better choices.

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.

If you are skipping ahead. The optimization and regularization knobs here carry almost unchanged into Chapters 12 to 14; what changes is how context is represented and propagated, not the core loss-minimization discipline.

Where we head next. Chapter 12 keeps the same training and audit discipline but shifts the modeling axis from spatial locality to temporal state and sequence dependence.

12 Introduction to Recurrent Neural Networks

Chapter 11 showed how architectural bias (convolution/pooling) improves sample efficiency while preserving the same optimization loop. This chapter keeps that loop but changes the axis from space to time: sequence models need memory so present predictions can depend on prior inputs. A useful continuity is *parameter sharing*: CNNs reuse one filter across spatial locations, while recurrent neural networks (RNNs) reuse one transition rule across time steps. Figure 1 flags this as the sequential branch of the neural strand.

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 a simple (ungated) RNN cell.
- Recognize practical stabilization techniques (gradient clipping, gating, normalization) that motivate later LSTM/Transformer chapters.

Design motif

Add recurrence, then train it by unrolling time and reusing the backprop machinery from Chapter 7.

The statistical learning chapters (Chapters 3 to 4) treated learning as an engineering loop: choose a model class, define a loss, optimize it, and then audit generalization. The feedforward neural chapters then expanded the model class from linear predictors to multilayer networks (MLPs, RBF networks, and CNNs), but the basic data interface stayed the same: each example is treated as a standalone input–output pair.

Sequence problems break that interface. When order matters, the model needs a state that can carry information forward, and the training loop must account for the fact that the same parameters are reused again and again across time.

12.1 Motivation for Recurrent Neural Networks

Most of the models we have built so far make a quiet assumption about the data: examples are independent and identically distributed (i.i.d.). If you shuffle the dataset, nothing essential changes. That assumption is fine for many supervised tasks on static inputs (tabular rows, images), but it breaks for sequences.

In time series, yesterday changes what you expect at the next step. In language, the same words in a different order can mean something else entirely, and the next word depends on the whole prefix. You can see that in predictive text: “I want to buy …” pulls you toward different completions than “Write a book about Teddy …” even before you specify what you are buying or writing about. In control, current action depends on prior state; in speech, a phoneme

depends on the surrounding phonemes.

One brute-force workaround is to build a fixed history window and feed it to a standard MLP: at time t, concatenate $\mathbf{x}_t, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_{t-k}$ and ask the network to predict y_t. That can work for short memory, but it scales poorly as k grows, and it does not reuse parameters in a way that reflects the repeated structure of "the same kind of step happens again and again".

Recurrent neural networks make the smallest structural change that addresses the bottleneck: add a *state* that is updated each step, and reuse the same update rule across time. The rest of this chapter is about making that idea precise, training it reliably, and recognizing the failure modes (especially gradient flow) that motivate gating and attention.

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.

Terminology (steps and representations). Throughout this chapter, t indexes *steps* in a sequence. In a time series, $\mathbf{x}_t$ can be a measured feature vector. In other domains the raw observation at step t can be discrete (a symbol from

a finite set) or structured (a set of features), but it is always converted into a numeric input vector $\mathbf{x}_t$ before it enters the recurrence. We keep the derivations in $\mathbf{x}_t$ so the same math applies across domains; Chapter 13 makes the "discrete symbols $\rightarrow$ vectors" story concrete for text.

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 order and history, 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. Chapter 13 then supplies the representation tools and evaluation lens commonly paired with language models, and Chapter 14 revisits sequence modeling without recurrence.

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.

What recurrence buys you. A feedforward network reacts to the current input $\mathbf{x}_t$ unless you deliberately pack history into the input features. An RNN adds a state $\mathbf{h}_t$ that carries information forward, so predictions can depend on order and on longer histories.

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.

Roadmap. We start with the state-update picture, then unroll it to derive BPTT. From there we look at what breaks on long sequences (vanishing/exploding gradients) and the practical fixes (truncation, clipping, normalization, and gated cells), and we end with a short bridge to attention.

12.4 Recap: Feedforward Building Blocks

Simple RNN at a glance. A simple (ungated) RNN is trained the same way as our earlier networks: pick a loss, compute gradients, and update parameters. For sequence prediction the default loss is cross-entropy accumulated over time, with state updates of the form $\mathbf{h}_t = f(\mathbf{x}_t W_{xh} + \mathbf{h}_{t-1} W_{hh} + b_h)$ and a per-step predictive distribution $p_\theta(y_t \mid \mathbf{h}_t)$. The main design knobs are the state size and how far you backpropagate through time (full BPTT versus a truncation window). In practice, most failures are either gradient blow-up/decay on long sequences or a train/inference mismatch when teacher forcing is used without a decoding strategy that matches it.

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 52 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.

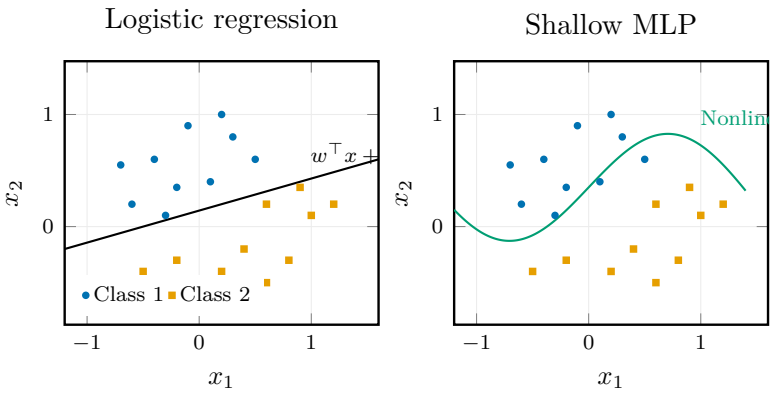

Figure 52: Decision boundaries for logistic regression (left) versus a shallow MLP (right). Linear models carve a single hyperplane, whereas hidden units can warp the boundary to follow non-convex manifolds such as the moons dataset.

Figure 54 is the canonical unrolling view used to interpret shared-parameter BPTT updates.

Finally, Figure 53 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 these when tuning sequence models, where overfitting appears as a divergence between per-step training and validation likelihood (per-token in text models).

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.

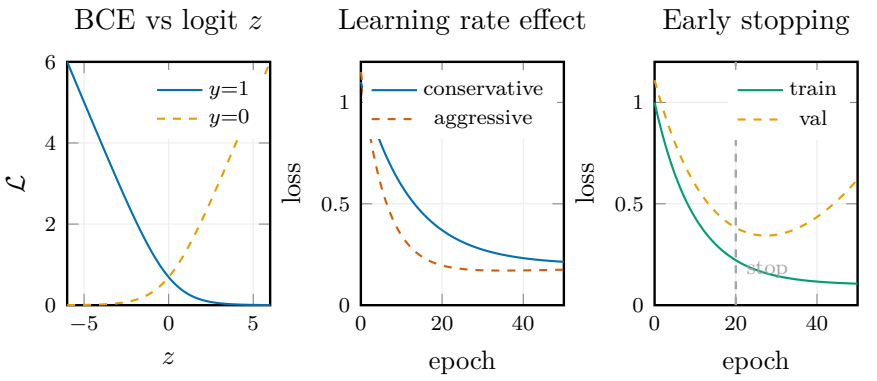

Figure 53: Binary cross-entropy geometry (left), effect of learning-rate schedules on loss (middle), and the typical training/validation divergence that motivates early stopping (right).

LayerNorm and residual RNN tips

Layer Normalization (Ba et al., 2016) stabilizes recurrent dynamics by normalizing each hidden vector $\mathbf{h}_t$ 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 and stable recurrence

Chapter 10 covered Hopfield networks as an associative memory: a recurrent system with symmetric weights and an energy function that decreases under the update rule, so the dynamics settle into stored patterns (attractors). That is a different use of recurrence than the one in this chapter.

Here we use recurrence to model *ordered* data. The goal is not to converge to a fixed point, but to carry a useful state forward so predictions at time t can depend on earlier inputs. The shared theme is that dynamics matter; the diagnostics differ (energy trace and overlap for Hopfield, versus likelihood/perplexity and gradient health for sequence models).

12.5 Input–output configurations and mathematical formulation

RNNs appear in a few standard input–output shapes. In a many-to-one task (e.g., sentiment), the model consumes $\mathbf{x}_{1:T}$ and emits one label from the final state. In many-to-many tasks (e.g., tagging), it emits an output at every step. In one-to-many settings (conditional generation), it starts from a conditioning signal (an initial state or an encoder output) and then generates autoregressively. Bidirectional encoders run the recurrence forward and backward and concatenate states when you want both left and right history during labeling, or when you want a richer encoder representation for a decoder.

Consider an input sequence $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T\}$, where each $\mathbf{x}_t \in \mathbb{R}^d$. The RNN computes hidden states $\{\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_T\}$ and outputs $\{\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_T\}$ as follows:

$$\mathbf{h}_0 = \mathbf{0} \quad \text{(initial hidden state)} \tag{12.4}$$

$$\mathbf{h}_t = f(\mathbf{x}_t W_{xh} + \mathbf{h}_{t-1} W_{hh} + b_h), \quad t = 1, \ldots, T \tag{12.5}$$

$$\mathbf{y}_t = g(\mathbf{h}_t W_{hy} + b_y), \quad t = 1, \ldots, T \tag{12.6}$$

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 $L = \sum_{b,t} M_{b,t}\, \mathrm{CE}(\hat{y}_{b,t}, y_{b,t}) / \sum_{b,t} M_{b,t}$, where $\mathrm{CE}(\cdot,\cdot)$ is the cross-entropy loss. Masks preview the padding/causal masks detailed in Chapter 14.

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).

Numeric check: shared weights mean summed gradients

It helps to see the "sum across time" in one tiny example. Consider a scalar RNN with identity activation,

$$h_t = W_{hh}h_{t-1} + W_{xh}x_t, \qquad y_t = h_t,$$

and a two-step squared-error objective $L = \frac{1}{2}\big[(y_1 - t_1)^2 + (y_2 - t_2)^2\big]$. Pick $h_0 = 1$, $W_{hh} = 0.5$, $W_{xh} = 1$, $(x_1, x_2) = (1, 2)$, and $(t_1, t_2) = (1, 2)$. The forward pass gives

$$h_1 = 1.5, \quad h_2 = 2.75, \quad e_1 = y_1 - t_1 = 0.5, \quad e_2 = y_2 - t_2 = 0.75.$$

Because W_{hh} is reused, its gradient receives contributions from *both* steps:

$$\frac{\partial L}{\partial W_{hh}} = e_1\frac{\partial h_1}{\partial W_{hh}} + e_2\frac{\partial h_2}{\partial W_{hh}} = e_1 h_0 + e_2\big(h_1 + W_{hh}h_0\big) = 2.0.$$

The second term is the one students often miss: changing W_{hh} changes h_1, which then changes h_2. A finite-difference check matches the value above.

12.6 Mathematical Formulation of a Simple RNN Cell

Consider a simple RNN cell with the following update equations:

$$\mathbf{h}_t = \sigma_h\left(\mathbf{x}_t\mathbf{W}_{xh} + \mathbf{h}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h\right), \tag{12.7}$$

$$\mathbf{y}_t = \sigma_y\left(\mathbf{h}_t\mathbf{W}_{hy} + \mathbf{b}_y\right). \tag{12.8}$$

12.7 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:

$$\mathcal{L} = -\sum_i y_i \log \hat{y}_i, \tag{12.9}$$

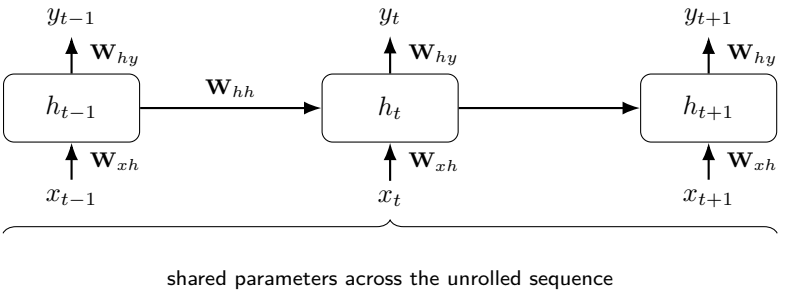

Figure 54: Unrolling an RNN reveals repeated application of the same parameters across time steps. This view motivates backpropagation through time (BPTT), which accumulates gradients through every copy before updating the shared weights.

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.

For 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 produces outputs $\hat{y}_t$ and the loss accumulates over time; gradients are computed via BPTT on the unrolled graph (see the unrolling discussion in Section 12.6 and the flow in Figure 55).

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).

Simple (ungated) 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 prefer LayerNorm over BatchNorm inside recurrence 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).

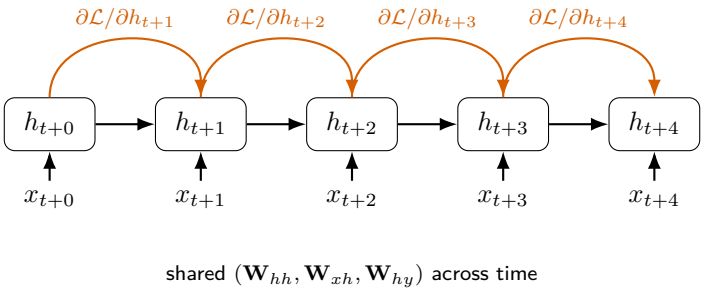

Figure 55: Backpropagation through time (BPTT): unrolled forward pass (black) and backward gradients (pink) through time.

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 $\operatorname{diag}\big(f'(\mathbf{a}_t)\big) \in \mathbb{R}^{d_h \times d_h}$, 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 56 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, simple (ungated) RNNs often struggle to preserve dependencies beyond roughly 5–10 steps in language settings, 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:

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial W}. \tag{12.11}$$

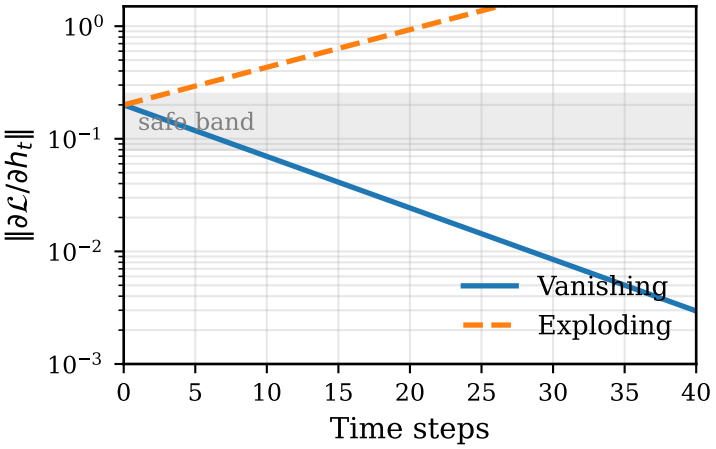

Figure 56: Vanishing (blue) versus exploding (orange) gradients on a log scale. The gray strip highlights the stability band; the inset reminds readers that repeated Jacobian products either shrink gradients (thin blue arrows) or amplify them (thick orange arrows).

Because of parameter sharing, the same W influences multiple time steps, and the total gradient is the sum over these contributions.

12.8 Stabilizing Recurrent Training

Gradient clipping. A practical safeguard is to clip the global norm of the gradient when it exceeds a threshold. Figure 57 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).

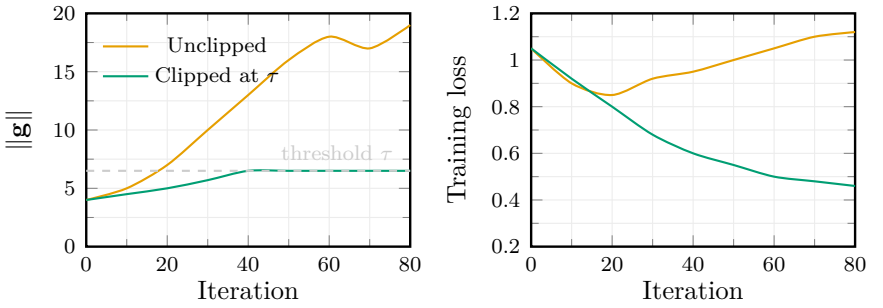

Figure 57: Gradient norms (left) explode without clipping (orange) but remain bounded when the global norm is clipped at tau (green). Training loss (right) stabilizes as a result.

Stability defaults in practice

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 a simple RNN. If your task needs dependencies longer than a handful of steps, do not over-tune an ungated RNN. Start with clipping and a sensible truncation window, but move to GRU/LSTM once long-range information vanishes.

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 58 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).

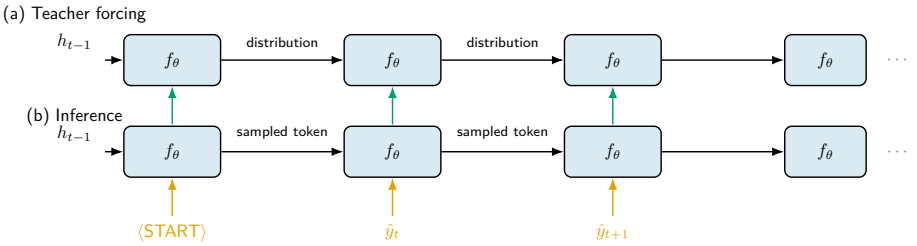

Figure 58: Teacher forcing vs. inference in a sequence-to-sequence decoder. Gold arrows show supervised targets; orange arrows highlight autoregressive feedback that motivates scheduled sampling.

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 59 and 60 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 $\mathbf{h}_{t-1}$ and a candidate $\tilde{\mathbf{h}}_t$ and decide how much past information 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 $\mathbf{h}_t = o_t \odot \tanh(c_t)$. In a GRU, the hidden state updates as $\mathbf{h}_t = (1 - z_t) \odot \mathbf{h}_{t-1} + z_t \odot \tilde{\mathbf{h}}_t$, with the reset gate r_t shaping the candidate computation. In both figures, $\mathbf{x}_t$ denotes the input at time t, and $\mathbf{h}_{t-1}$ / c_{t-1} denote the carried state(s).

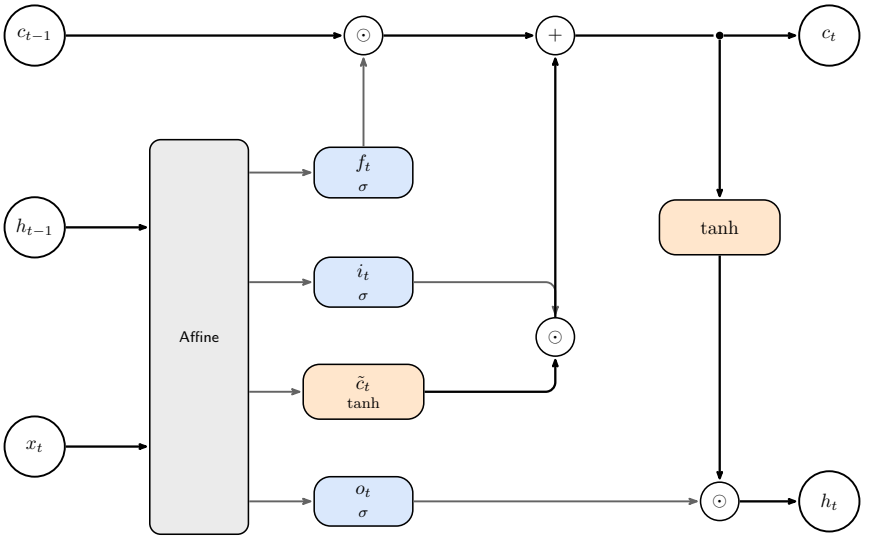

Figure 59: Long Short-Term Memory (LSTM) cell (Hochreiter and Schmidhuber, 1997; Gers et al., 2000).

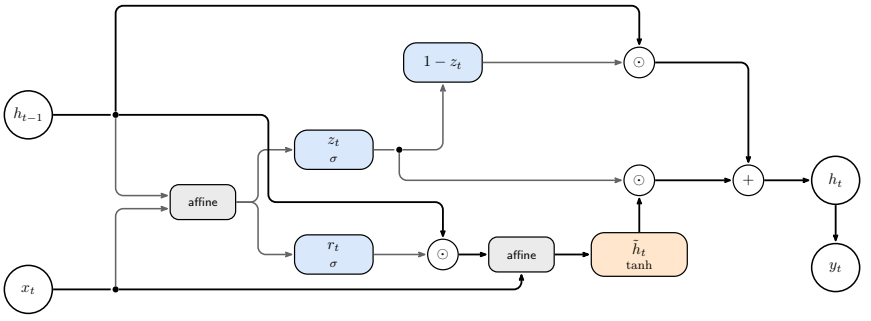

Figure 60: Gated Recurrent Unit (GRU) cell (Cho et al., 2014).

Simple RNN vs. GRU vs. LSTM

A simple (ungated) RNN has one state $\mathbf{h}_t$ and is the most parameter-efficient, but it is also the easiest to break on long sequences (vanishing/exploding gradients). A GRU adds gates that decide how much past state to keep versus overwrite, and it is often the best default when you want something robust without the full LSTM machinery. An LSTM maintains a separate cell state c_t with input/forget/output gates; it is heavier, but it tends to be the most forgiving when you truly need long-range retention.

Practical default: if you are unsure, start with a GRU (plus clipping and masking) and only simplify to a simple RNN for short-memory problems.

Implementation checklist: masks, clipping, pitfalls

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)
    # CE = per-step cross-entropy (e.g., reduction='none').
    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.

Common pitfalls.

- Pads leaking into the loss (mask missing or applied after reduction).
- Hidden state carried across unrelated sequences (forgetting to reset or detach).
- No clipping on long sequences (exploding gradients is the first failure mode).
- BatchNorm inside recurrence (prefer LayerNorm for sequence models).
- Teacher forcing at training time without thinking through inference-time decoding.
- Dropout applied independently at each step when you intended variational/recurrent dropout.

12.9 From recurrent state to attention

Attention mechanisms. Even with gating, long sequences can challenge fixed-size hidden states. Attention augments the decoder with a content-based lookup into the encoder states, as visualized in Figure 61. Bright entries corre-

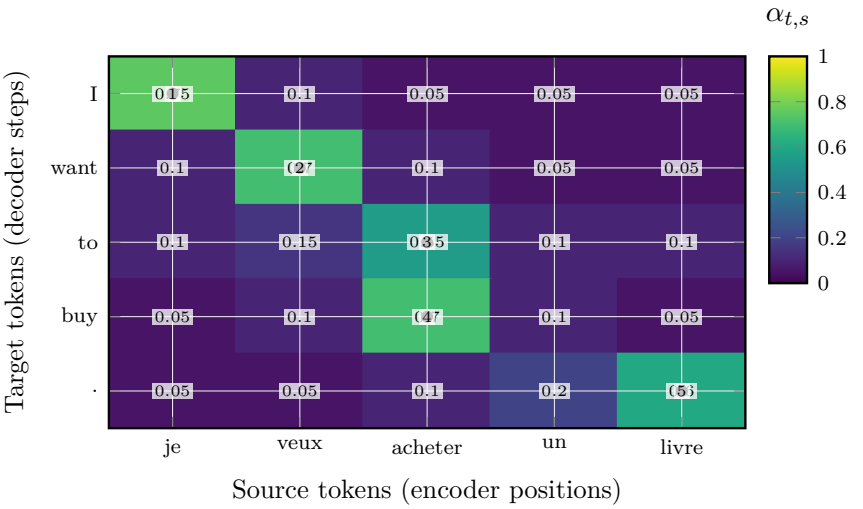

Figure 61: Attention heatmap for a translation model. Rows are target tokens (decoder steps) and columns are source tokens (encoder positions). Each cell is an attention weight; the dot in each row marks the source position receiving the most attention.

spond to encoder positions that most influence each generated token.

Historical note: from gating to attention

Gating (LSTM/GRU) made long-range gradient flow workable by creating additive memory paths. Attention takes a different route: rather than forcing all history into a single state, it learns a soft retrieval rule over past representations. This shift explains why modern sequence models often prefer attention when long-range dependencies and parallel training matter most, while gated RNNs remain competitive for streaming and low-latency settings.

12.10 Wrapping Up the Derivations

One canonical sequence task is *next-step prediction*: use the history up to step t to predict a target at step $t + 1$. The target can be continuous (forecasting), categorical (a label at each step), or discrete from a finite set (symbols/words). The mechanics are the same: define a per-step predictive distribution (or regression output), accumulate a loss over time, and train by BPTT on the unrolled

computation graph.

12.10.1 Optional: language modeling as next-step prediction (preview)

Language modeling is the discrete-symbol case: predict the next token from the preceding prefix.

Recall that the goal is to estimate the conditional probability of a word w_t given the sequence of previous words $w_1, w_2, \ldots, w_{t-1}$:

$$P(w_t \mid w_1, w_2, \ldots, w_{t-1}). \tag{12.12}$$

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.

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t; \theta), \tag{12.13}$$

$$P(w_{t+1} \mid w_1, \ldots, w_t) = g(\mathbf{h}_t; \theta), \tag{12.14}$$

where $\mathbf{x}_t$ is the input representation (e.g., a word embedding) of the token 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, $\mathbf{h}_t$ is a deterministic summary of the entire prefix $(w_1, \ldots, w_t)$:

$$\mathbf{h}_t = f(f(\cdots f(\mathbf{h}_0, \mathbf{x}_1), \ldots), \mathbf{x}_t).$$

What the model can remember (and what it must forget) is governed by the state dimension, the update nonlinearity, and the training objective. That is exactly why gradient flow and stabilization tricks matter in practice.

Training objective (negative log-likelihood). Given a training sequence $(w_1, w_2, \ldots, w_T)$, we typically train by minimizing the negative log-likelihood (NLL) (equivalently, maximizing the likelihood). With the next-token conven-

tion, the loss is

$$\mathcal{L}(\theta) = -\sum_{t=1}^{T-1} \log P(w_{t+1} \mid w_1, \ldots, w_t; \theta). \tag{12.15}$$

This is just cross-entropy over the vocabulary at each step: the model is rewarded for assigning high probability to the observed next token. A common reporting metric is *perplexity*, defined as exp(average NLL per token); lower perplexity means the model assigns higher probability to the observed text. If you are not working with text, you can ignore perplexity—the underlying idea is simply "negative log-likelihood per step." Chapter 13 returns to this objective in a more application-facing way (tokenization, embeddings, and audit considerations).

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.

Input representations. The input $\mathbf{x}_t$ is a numeric vector. In language modeling and other discrete-token problems, it is typically a token embedding produced by a learned lookup table. Chapter 13 covers where those vectors come from (training objectives, evaluation, and audit considerations).

12.10.2 Worked example: sentiment as a many-to-one decision

To make the sequence viewpoint concrete, consider a short review such as: *"This place is great."* In a sentiment classifier, we map the token sequence $(w_1, \ldots, w_T)$ to a single label $y \in \{0, 1\}$ (negative/positive). The recurrence consumes numeric inputs $\mathbf{x}_t$ (one per token) and produces a final state $\mathbf{h}_T$ that summarizes the sentence:

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t; \theta), \qquad \hat{y} = \sigma(\mathbf{w}^\top \mathbf{h}_T + b), \tag{12.16}$$

where $\sigma(\cdot)$ is the logistic sigmoid and $\hat{y}$ is the predicted probability of the positive class.

This setup highlights a subtle but important dependency between *representation* and *generalization*. If tokens are treated as unrelated IDs (e.g., purely one-hot indicators), then "great" and "awesome" are orthogonal inputs: the model has no built-in reason to transfer evidence across similar words unless the training objective supplies a geometry that makes them nearby. In Chapter 13, we return to this same kind of task and show how learning word vectors from neighboring words gives the model a meaningful input space before any sequence model is applied.

From sequence data to a next-step model (a minimal checklist)

A minimal sequence-model run looks like this. First decide how each step becomes a numeric input vector $\mathbf{x}_t$ (measured features, engineered features, or learned embeddings). Next decide the prediction target shape (many-to-one label, many-to-many labeling, or next-step prediction), then pick the recurrent cell f (simple RNN, GRU, or LSTM) and the output map g. Train with a per-step loss summed over time using truncated BPTT and correct padding masks. Finally, monitor validation loss and gradient norms; clipping is the first safety valve when sequences get long.

It is the same ERM loop as earlier chapters; the new wrinkle is that the loss (and therefore the gradients) accumulates across time as well as across batches. In Chapter 13, we specialize this checklist to text (discrete symbols, embeddings, next-token loss, and perplexity).

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; see Appendix D for the cross-chapter symbol index.

Optional refresher (padding masks and encoders). Variable-length sequences are almost always padded to a common length for batching. The padding is not data, so you must carry a mask through the loss (and through attention later) so pads contribute *zero* training signal. For encoder–decoder intuition, it helps to remember that many sequence models compress a history into a state (or a set of states) and then produce outputs from that representation; Chapter 14 revisits this idea with attention rather than recurrence.

Key takeaways

- Sequence losses add across time; shared parameters mean gradients accumulate across unrolled steps (BPTT).
- Truncated BPTT and padding masks are the engineering tools that make training feasible on long and variable-length sequences.
- Stability tools recur across architectures: clipping, gating (GRU/LSTM), and normalization; attention offers an alternative route to long-range dependencies.
- For text, the sequence is discrete and inputs are embeddings; Chapter 13 makes that concrete and introduces next-token cross-entropy and perplexity as a domain-specific diagnostic.

A good checkpoint.

- Explain BPTT, truncation, and masking for variable-length sequences.
- Describe why gates help (information and gradient flow) and when attention is a practical alternative.
- If you care about language modeling, relate cross-entropy, log-likelihood, and perplexity after reading Chapter 13.

Common pitfalls.

- Train/inference mismatch (teacher forcing without a decoding strategy that reflects it).
- Unstable gradients on long sequences (no clipping, poor initialization, or overly large learning rates).
- Incorrect padding masks that leak future tokens or contaminate the loss.

Exercises and lab ideas

- Train a many-to-one sentiment classifier; plot gradient norms with and without clipping.
- Train a small time-series predictor; compare full BPTT vs. truncated BPTT windows and quantify the stability trade-off.
- Implement a many-to-many tagging model; verify that padding masks do not affect the loss or gradients.
- Optional preview: train a small LSTM next-token model; compare perplexity with/without weight tying and scheduled sampling (then revisit in Chapter 13).
- 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.

If you are skipping ahead. Keep the loss/perplexity diagnostics and masking discipline from this chapter. Chapter 14 changes the architecture, but the same evaluation traps and data-leakage risks remain.

Where we head next. Chapter 13 translates this sequence viewpoint into representation tooling: embedding objectives, neighborhood/analogy diagnostics, and deployment audits. Keep the sentiment example (Section 12.10.2) in mind: with learned token geometry, the same many-to-one task generalizes better than with raw IDs. Chapter 14 then changes architecture again, replacing recurrence with attention while preserving the same evaluation and masking discipline.

13 Neural Network Applications in Natural Language Processing

Chapter 12 framed language as prediction under context: the model must assign probability to the next token from a compressed state. That framing makes

the bottleneck obvious. Tokens are discrete identifiers, while the model is a continuous function; if the input geometry is unhelpful, the network cannot generalize systematically from one string to another.

This chapter supplies that geometry. We build word embeddings and the objectives that train them, then show how to sanity-check the resulting space (neighbors, analogies, and simple audits). Those same representation primitives become inputs to both recurrent and attention-based models; Chapter 14 reuses them when recurrence is replaced by attention.

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.

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.

13.1 Context and Motivation

In this chapter, we focus on neural methods for natural language processing (NLP) through the lens of representation learning. The recurring question from Chapter 12 is practical: what input space makes sequence prediction stable and data efficient? The answer is to learn vectors whose geometry captures contextual similarity.

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.

How this chapter is organized. We start with the linguistic idea (distributional semantics) and immediately turn it into an engineering object: a lookup table of vectors that make prediction from nearby words easier. From there we build the two Word2Vec objectives (CBOW and skip-gram), show why full softmax is expensive, and introduce negative sampling as the practical workaround. We then discuss how to sanity-check an embedding space (neighbors and analogies) and close with what can go wrong in deployment (bias, leakage, and basic audits).

Notation. We use V for vocabulary size, d for embedding dimension, $\mathbf{x}_t$ for one-hot token indicators, and $\mathbf{e}_t$ for learned embedding vectors. When symbols collide with earlier chapters (e.g., d as data dimension vs. embedding size), the local meaning is authoritative; Appendix D gives the global index.

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 begins as an index (equivalently, a one-hot indicator). 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.

13.2 Warm-up: from symbols to vectors

One-hot vectors make discrete tokens compatible with linear algebra, but they do not express *similarity*: synonyms are orthogonal, and a model that treats each basis element independently has no reason to generalize from "great" to "awesome" without seeing both during training.

One-hot encoding (baseline). Fix a dictionary of size V. A token at position t can be represented as a one-hot row indicator $\mathbf{x}_t \in \{0, 1\}^{1 \times V}$ with a single

1. This guarantees uniqueness, but it makes every pair of distinct words equally far apart.

Why one-hot is not enough (two quick failures). Synonyms. A sentiment model trained to treat "great" as positive has no immediate reason to treat "awesome" as positive if the two words are orthogonal basis vectors.

Document similarity. "I enjoyed talking to the monarchs" and "I loved conversing with the royals" express the same meaning, but a bag-of-words similarity score stays low if it cannot recognize that *monarchs* and *royals* occupy similar roles.

This is the same friction point you saw in Section 12.10.2: the sequence model can only be as systematic as its input geometry.

Embedding lookup as learned features. Instead of using the identity basis, we learn an embedding table $\mathbf{W} \in \mathbb{R}^{V \times d}$ whose rows are trainable parameters. A one-hot indicator becomes a row lookup:

$$\mathbf{e}_t = \mathbf{x}_t \mathbf{W} \in \mathbb{R}^{1 \times d}.$$

This is the same linear algebra as one-hot, but now the coordinates are *learned* so that similar words end up nearby when that helps prediction.

A useful mental model is to imagine that each word is a *graded bundle of attributes*. If we could write those attributes down explicitly, each word would already be a vector:

Table 5 gives a concrete intuition for mapping symbolic tokens into graded feature vectors.

Numeric check: why the analogy arithmetic can work

Using the toy feature vectors in Table 5, the analogy is literal:

$$\mathbf{v}(\text{king}) - \mathbf{v}(\text{man}) + \mathbf{v}(\text{woman}) = \mathbf{v}(\text{queen}).$$

Real embeddings are not hand-designed, so the equality is only approximate, but the geometry is the same: a shared "relation direction" can show up as an approximately consistent offset across many word pairs. In this toy table, you can verify the identity directly from the entries.

Table 5: 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. Treat this as an intuition pump: real embedding coordinates are learned from data rather than hand-assigned.

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

Of course, hand-crafting features does not scale to large vocabularies, and it bakes in subjective choices. The central move of modern embeddings is to let the *training objective* discover whatever coordinates best support prediction from context.

Subword tokenization and OOV handling. Practical systems rarely operate on raw word types alone. To reduce vocabulary size and handle out-of-vocabulary (OOV) forms, 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 re-composed 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.

13.3 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.

13.4 Contextual Meaning and Feature Extraction

The practical takeaway is that repeated context patterns reveal latent word properties. In the *pretty* example, co-occurrence with evaluative adjectives points to an intensifier sense, while co-occurrence with visual nouns points to an aesthetic sense.

This is why modern embedding models treat corpus co-occurrence as supervision.

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.

13.5 Word2Vec at a glance

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 vectors by solving local prediction tasks rather than from hand-labeled semantic features.

13.5.1 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*.

13.5.2 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.

This first pass is conceptual. The next sections pin down the exact parameterization and training objectives used in practice.

13.6 From lookup to objective: compact derivation path

With the conceptual picture fixed, we now run one continuous derivation path from lookup mechanics to CBOW/skip-gram training objectives. Let 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).*

Embedding lookup as a tiny network (shapes). Fix a vocabulary of size V and an embedding dimension d (e.g., $V = 10{,}000$, $d = 300$). A token is an index $i \in \{1, \ldots, V\}$, represented as a one-hot row vector $\mathbf{x} \in \{0, 1\}^{1 \times V}$. The embedding table $\mathbf{W} \in \mathbb{R}^{V \times d}$ is trainable, and the embedding for the token is the row lookup

$$\mathbf{h} = \mathbf{xW} \in \mathbb{R}^{1 \times d}. \tag{13.1}$$

Because $\mathbf{x}$ is one-hot, this multiplication selects exactly one row of $\mathbf{W}$; that is the whole "embedding lookup" operation.

To turn the lookup into a prediction task (CBOW or skip-gram), we add output-side vectors $\mathbf{W}_{\text{out}} \in \mathbb{R}^{d \times V}$ and compute logits

$$\mathbf{z} = \mathbf{hW}_{\text{out}} \in \mathbb{R}^{1 \times V}. \tag{13.2}$$

These logits are passed through a softmax function to produce a probability distribution over the vocabulary:

$$\hat{\mathbf{y}}_j = \frac{\exp(z_j)}{\sum_{k=1}^{V} \exp(z_k)}, \quad j = 1, \ldots, V. \tag{13.3}$$

Training then minimizes cross-entropy (equivalently, negative log-likelihood)

against the one-hot target $\mathbf{y}$:

$$\mathcal{L} = -\sum_{j=1}^{V} y_j \log \hat{y}_j. \tag{13.4}$$

During learning, $\mathbf{W}$ and $\mathbf{W}_{\text{out}}$ update so that words that share contexts end up with similar vectors.

Context window and sequential input. Reusing the same toy phrase, let the context window size be 4 around the target token. To predict "automatic" in "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.

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.

13.7 Word2Vec objectives in detail

With notation fixed, we now write the two Word2Vec objectives in full form: Continuous Bag of Words (CBOW) and skip-gram.

13.7.1 CBOW objective (detailed form)

In the detailed CBOW form, the model predicts center token w_t from its window context. For a sequence $w_1, w_2, \ldots, w_T$ and half-window size c, the context is $\{w_{t-c}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+c}\}$.

Each context token is encoded as a one-hot row in $\mathbb{R}^{1\times V}$, then mapped through $\mathbf{W} \in \mathbb{R}^{V\times d}$ to its dense 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 $\mathbf{W}_{\text{out}}$:

$$\mathbf{h} = \frac{1}{2c} \sum_{\substack{-c\le j\le c \\ j\neq 0}} \mathbf{x}_{t+j}\mathbf{W} \tag{13.5}$$

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:

$$P(w_t \mid \text{context}) = \frac{\exp(\mathbf{h}\,\mathbf{u}_{w_t})}{\sum_{w=1}^{V} \exp(\mathbf{h}\,\mathbf{u}_w)} \tag{13.6}$$

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 maximizes corpus log-likelihood, updating both $\mathbf{W}$ and output vectors $\mathbf{u}_w$. The rows of $\mathbf{W}$ are then used as the learned embeddings.

Key Insight: Multiplying one-hot input $\mathbf{x}_{t+j}$ by $\mathbf{W}$ performs a row lookup. The embedding matrix therefore behaves as a trainable feature table.

13.7.2 skip-gram objective (detailed form)

The detailed skip-gram form reverses CBOW: use center word w_t to predict each context token w_{t+j} in window c:

$$\prod_{\substack{-c \le j \le c \\ j \neq 0}} P(w_{t+j} \mid w_t) \tag{13.7}$$

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 $\mathbf{W} \in \mathbb{R}^{V \times d}$:

$$\mathbf{h} = \mathbf{x}_t \mathbf{W} \tag{13.8}$$

Each context word w_{t+j} is predicted by applying a softmax over the output vectors (again using the output-embedding matrix $\mathbf{W}_{\text{out}}$):

$$P(w_{t+j} \mid w_t) = \frac{\exp\big(\mathbf{h}\,\mathbf{u}_{w_{t+j}}\big)}{\sum_{w=1}^{V} \exp(\mathbf{h}\,\mathbf{u}_w)} \tag{13.9}$$

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: Skip-gram pushes words with similar neighborhoods toward similar embeddings.

13.7.3 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 (13.6) and (13.9) involves summing exponentials over all vocabulary words:

$$Z = \sum_{w=1}^{V} \exp(\mathbf{h}\,\mathbf{u}_w) \tag{13.10}$$

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 $P_n(w) \propto f(w)^{0.75}$, learning rate 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.

13.8 Efficient Training of Word Embeddings: Hierarchical Softmax and Negative Sampling

We now expand the two approximations introduced above—hierarchical softmax and negative sampling—and use them throughout the rest of the section.

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:

$$p(D = 1 \mid w, c) = \sigma(\mathbf{v}_w \, \mathbf{v}'_c) \tag{13.11}$$

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:

$$\log \sigma(\mathbf{v}_w \, \mathbf{v}'_c) + \sum_{i=1}^{k} \log \sigma\big(- \mathbf{v}_w \, \mathbf{v}'_{c'_i}\big) \tag{13.12}$$

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 (13.12). 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.

13.9 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.

13.10 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-

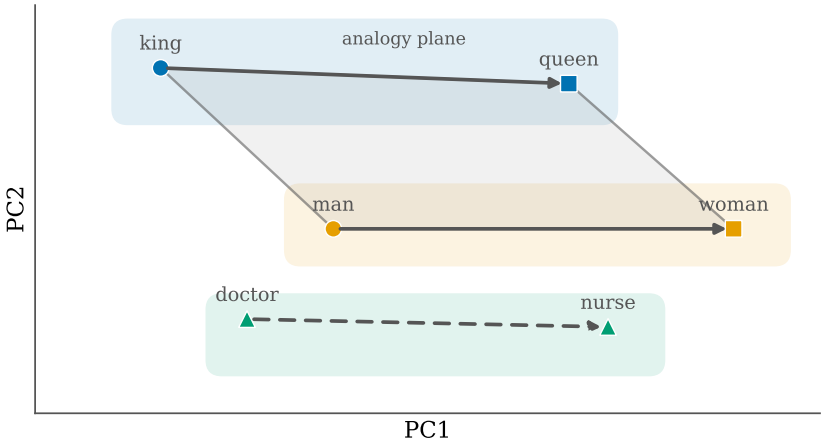

Figure 62: Analogy geometry in embedding space. The offset "v(king) - v(man) + v(woman) approx v(queen)" forms a parallelogram; a similar gender direction shifts "doctor" toward "nurse." Solid and dashed vectors highlight the shared relational direction. Points are shown after a 2D principal component analysis (PCA) projection, so directions are approximate.

ation.

Visualization. Projecting the learned vectors onto two principal components typically reveals well-separated semantic clusters. Figure 62 highlights how gendered titles, fruits, and locations occupy distinct regions, reinforcing that co-occurrence-driven training captures rich lexical structure.

13.10.1 Modeling Co-occurrence Probabilities

We start by considering the conditional probability of observing a context word w_k given a target word w_i:

$$P(k|i) = \frac{X_{ik}}{X_i}. \tag{13.13}$$

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:

$$P(k|i) \propto \exp\left(\mathbf{v}_i \, \mathbf{u}_k\right). \tag{13.14}$$

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

$$\log P(k|i) \approx \mathbf{v}_i \, \mathbf{u}_k + b_i + b_k, \tag{13.15}$$

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,

$$\log X_{ik} - \log X_i = \log \frac{X_{ik}}{X_i} = \log P(k|i) \tag{13.16}$$

$$\approx \mathbf{v}_i \, \mathbf{u}_k + b_i + b_k. \tag{13.17}$$

This equation suggests that the log co-occurrence counts can be approximated by a bilinear form plus biases.

13.10.2 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:

$$J = \sum_{i,k} f(X_{ik}) \left(\mathbf{v}_i \, \mathbf{u}_k + b_i + b_k - \log X_{ik}\right)^2, \tag{13.18}$$

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:

$$f(x) = \begin{cases} \left(\frac{x}{x_{\max}}\right)^{\alpha} & \text{if } x < x_{\max}, \\ 1 & \text{otherwise,} \end{cases} \tag{13.19}$$

where $\alpha \in (0, 1)$ and $x_{\max}$ is a cutoff parameter.

13.11 Finalizing the Word Embedding Derivations

In the previous sections, we explored the formulation of word embeddings through co-occurrence statistics and matrix factorization approaches. This section closes the derivation arc and clarifies 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}:

$$\mathbf{v}_i \, \mathbf{u}_k + b_i + b_k = \log X_{ik}, \tag{13.20}$$

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

$$\mathbf{v}_i \, \mathbf{u}_k = \log X_{ik} - b_i - b_k. \tag{13.21}$$

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

$$J = \sum_{i=1}^{V} \sum_{k=1}^{V} f(X_{ik}) \left(\mathbf{v}_i \, \mathbf{u}_k + b_i + b_k - \log X_{ik}\right)^2, \tag{13.22}$$

where $f(x)$ is the weighting function that downweights rare (or extremely common) co-occurrences to improve robustness. GloVe uses the piecewise definition in (13.19), so very small counts contribute little to the loss while moderately frequent pairs still 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:

$$\log X \approx \mathbf{U}_r \mathbf{\Sigma}_r \mathbf{V}_r^\top, \tag{13.23}$$

where $\mathbf{U}_r \in \mathbb{R}^{V \times r}$ and $\mathbf{V}_r \in \mathbb{R}^{V \times r}$ contain the top-r singular vectors (for the desired embedding dimension $d = r$), and $\mathbf{\Sigma}_r \in \mathbb{R}^{r \times r}$ 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

$$\mathbf{v}_i = (\mathbf{U}_r)_i \mathbf{\Sigma}_r^{1/2}, \quad \mathbf{u}_k = (\mathbf{V}_r)_k \mathbf{\Sigma}_r^{1/2},$$

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.

Before moving from geometry to governance, keep the boundary clear: the sections above establish how embeddings are learned and why their vector arithmetic works. The next sections address whether that learned geometry is reliable and safe under real deployment constraints.

Risk & audit

- **Evaluation leakage:** similarity/analogy benchmarks can overlap training sources; keep a truly held-out evaluation set and treat "standard" datasets as potentially contaminated.
- **Tokenization debt:** preprocessing and vocabulary choices (case, subwords, cutoffs) change what the model can represent; version tokenizers and report them with results.
- **Frequency bias:** rare words get unstable vectors; audit neighborhoods by frequency and use subsampling/regularization so geometry is not just Zipf effects.
- **Social bias:** co-occurrence reflects social structure and stereotypes; probe for bias before using embeddings in decisions and document mitigations.
- **Privacy/memorization:** large corpora can contain sensitive strings; treat training data as a security boundary and audit downstream systems for memorization.

13.12 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. Digitized literature from the 19th century, for instance, over-represents colonial perspectives.
- **Language-specific bias:** Different languages and dialects encode dif-

ferent 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 There is no single fix, because the "bias direction" you measure depends on the corpus, the task, and the group definitions you care about. In practice, mitigations usually fall into three buckets: post-processing (identify a direction and project it out; e.g., Hard Debiasing by Bolukbasi et al. (2016)), data-side interventions (rebalance or augment so the corpus does not teach the same stereotype as strongly), and objective-side constraints (regularize training so certain associations are discouraged). The key is to treat mitigation as iterative: apply a change, then rerun the same probes and downstream slices you used to detect the issue.

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 (expected calibration error (ECE), reliability diagrams) overall and for key subgroups; severely miscalibrated models magnify harm when used for decision support.
- **Disaggregated evaluation:** Report accuracy, receiver operating characteristic / precision–recall (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.

13.13 Responsible deployment checklist

1. **Purpose & consent.** Document the use-case, decision stakes, and where humans remain in the loop; distinguish exploratory prototypes from production decision aids.
2. **Data lineage & licensing.** Track licenses for each corpus (newswire, Common Crawl, proprietary logs) and state whether downstream users may redistribute the embeddings or derived models.
3. **Privacy & security.** Scan corpora for personally identifiable information

(PII), redact when necessary, and restrict raw-data access. When embeddings leave the lab, accompany them with an acceptable-use policy and redaction guarantees.

4. **Monitoring.** Deploy subgroup-aware metrics, calibration checks, and toxicity filters in production; log drifts and institute retraining/rollback thresholds.

5. **Documentation.** Ship a short "model card" summarizing intended uses, failure modes, and evaluation data so downstream teams can reason about fit-for-purpose decisions.

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

This chapter has developed word-embedding models from co-occurrence statistics, including the role of bias terms and optimization tools such as singular value decomposition. It also emphasized a practical constraint: embeddings inherit social and cultural structure from training corpora, so bias diagnosis and mitigation are part of model quality, not an optional add-on.

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.

Minimum viable mastery.

- Explain how co-occurrence statistics induce geometry (dot products, cosine similarity, and linear offsets).
- Distinguish local-window objectives from global matrix factorization views and state when each is a good approximation.
- Identify at least one concrete bias test and one mitigation strategy, and articulate their limitations.

Common pitfalls.

- Treating nearest neighbors as meaning rather than distributional evidence (polysemy and domain shift).
- Over-interpreting analogy accuracy without controlling for frequency and evaluation set construction.
- Applying debiasing as a post-hoc patch while ignoring corpus composition and labeling practices.

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.

If you are skipping ahead. You can treat embeddings as feature vectors for any downstream model, but the audit mindset matters: log your corpus choices and evaluation splits so that later "fairness" or calibration conclusions have context.

Where we head next. Chapter 14 keeps the same token geometry but changes context formation: attention replaces recurrent state, so dependencies are queried directly. The same many-to-one decisions from Section 12.10.2 remain, but computational path and scaling behavior change. Keep the same audit lens as you read: tokenization choices, masking correctness, and evaluation protocol matter as much as architecture. After that, Chapter 15 pivots to soft computing (fuzzy logic and evolutionary ideas) as an alternative paradigm for reasoning under uncertainty.

14 Transformers: Attention-Based Sequence Modeling

Chapter 12 made the sequence problem explicit: stateful computation plus gradients that must flow across time. Chapter 13 then supplied the representation layer that makes those sequence objectives practical. This chapter keeps those representations but loosens the "one step at a time" constraint: instead of marching through time, we let each position look around and pull in what it needs through attention.

Learning Outcomes

After this chapter, you should be able to:

- Explain the encoder–decoder bottleneck and how attention fixes it in sequence-to-sequence problems.
- Write scaled dot-product attention and multi-head attention, and interpret them as weighted averages.
- Distinguish self-attention from cross-attention and describe where each appears in a Transformer.
- Explain positional encodings and masking (padding/causal) in training and decoding.
- Describe a Transformer block (residual paths, layer norm, FFN) and common objectives (MLM/CLM) and families (BERT/GPT/encoder–decoder).

Design motif

Make information flow explicit. Attention is a controlled mixing operation; masks enforce which interactions are allowed; residual paths and normalization keep optimization stable as models deepen and context windows grow.

Author's note: models, world models, and language models (an opinionated lens)

A *model* predicts outcomes from captured information. In the chapters so far, that "outcome" ranged from a class label, to a time-step forecast, to a retrieved memory pattern.

A *world model*, as I use the phrase here, is aspirational: it is the idea of a general-purpose internal model that can represent situations and dynamics well enough to support planning, counterplanning, and "what-if" reasoning. Whether we can build such systems reliably is part of the larger research program.

Language models are a simplified and very constrained training interface to this aspiration: at training time the next action is a token predicted from previous tokens. My opinion is that this setup can *suggest* a kind of internal world structure (because the output must remain internally and externally consistent to be useful), but it does not *guarantee* that a faithful world model has formed. Fluency is not proof of understanding; it is a behavior you still have to audit.

14.1 From encoder–decoder bottlenecks to attention

Sequence-to-sequence (seq2seq) problems have a natural story: read an input sequence and produce an output sequence. Translation is the canonical example. In the classical encoder–decoder picture, the encoder reads the source sentence and produces a representation; the decoder then uses that representation to generate the target sentence token by token.

We will keep the term *token* from Chapter 13: a token is a discrete symbol index that gets mapped to a vector. In text it might be a word or subword; in other sequence problems it can be any event ID you embed into a feature vector.

The practical bottleneck is the fixed-vector squeeze. If the decoder is only given one summary vector, it is forced to carry *everything* about the source through time, even when the next output token only depends on a small part of the input. Attention is the engineering fix: at each decoding step, compute a weighted average of the encoder states and let that mixture act as the context for the next prediction. In other words, the decoder is allowed to look back at the encoder's "memory" and ask which source positions matter *right now*.

Transformers (Vaswani et al., 2017) take that idea seriously and push it further. They remove recurrence and make "look around and mix" the core operation inside a layer. This allows parallel computation across positions and makes long-range interactions a first-class design choice rather than a side effect of how well information survives through many recurrent steps.

Seq2seq with attention (cross-attention). In an encoder–decoder RNN, the encoder produces a sequence of hidden states $\{\mathbf{h}_j\}_{j=1}^{S}$ from the input $\{\mathbf{x}_j\}$, and the decoder produces a sequence of states $\{\mathbf{s}_t\}_{t=1}^{T}$ while generating the output $\{y_t\}$. A convenient probabilistic view of translation is

$$p(\mathbf{y} \mid \mathbf{x}) = \prod_{t=1}^{T} p(y_t \mid y_{<t}, \mathbf{x})\,. \tag{14.1}$$

The "one-vector bottleneck" appears when the decoder is forced to rely on a single summary of the entire input. Attention replaces that with a *per-step context*:

$$e_{tj} = a(\mathbf{s}_{t-1}, \mathbf{h}_j), \tag{14.2}$$

$$\alpha_{tj} = \mathrm{softmax}_j(e_{tj}), \tag{14.3}$$

$$\mathbf{c}_t = \sum_{j=1}^{S} \alpha_{tj}\, \mathbf{h}_j. \tag{14.4}$$

Here $\mathrm{softmax}_j$ denotes a softmax over the source positions j, i.e., $\alpha_{tj} = \exp(e_{tj}) / \sum_{j'=1}^{S} \exp\big(e_{tj'}\big)$. The scoring function $a(\cdot,\cdot)$ can be a dot product, a bilinear form, or a small MLP, and $\mathbf{c}_t$ is the context the decoder uses when predicting y_t. This style of encoder–decoder attention traces back to early neural machine translation work (Bahdanau et al., 2015). That is the core move: different output positions can "look back" at different parts of the source. This is the bridge from encoder–decoder RNNs to Transformers: the attention math stays; what changes is that Transformers build the states $\mathbf{h}_j$ and $\mathbf{s}_t$ without recurrence.

14.2 Scaled Dot-Product Attention

Author's note: attention is a weighted average

RNNs and gated cells give you a way to carry *memory* forward, but memory is not the same thing as *relevance*. When you read a page, some words are glue and some words carry the meaning you need right now; your mind does not treat the whole history as equally important at every step. I like to read attention as the engineering version of that idea: a weighted average over candidate pieces of information. A query asks a question (what do I need?), keys advertise what each candidate is about (what do I contain?), and values carry the content that gets mixed. Similarity scores between queries and keys become nonnegative weights that sum to one; the output is the weighted sum of the value vectors.

A tiny analogy: keys, values, and weighted retrieval

Think of a tiny "database" with keys $\{70, 80\}$ and values $\{1000, 1500\}$. If your query is exactly 70, you retrieve 1000. If your query is 75, there is no exact match, so you can do a soft retrieval: score each key by a similarity such as

$$s_i = \frac{1}{|k_i - q| + \epsilon},$$

then normalize those scores so they sum to one and use them as weights on the values. Here $|70 - 75| = |80 - 75| = 5$, so $s_1 \approx s_2 \approx 0.2$, the normalized weights are $(0.5, 0.5)$, and the weighted average gives $0.5 \cdot 1000 + 0.5 \cdot 1500 = 1250$.

This is only an intuition pump: Transformers do not use scalar keys like "70." They learn vector keys and queries and score them in a learned feature space. But the weighted-average mechanism is exactly the same.

At the per-position level, the computation is

$$\mathbf{z}_i = \sum_j \alpha_{ij}\, \mathbf{v}_j, \qquad \alpha_{ij} \geq 0, \quad \sum_j \alpha_{ij} = 1,$$

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

$$\mathrm{Attn}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}. \tag{14.5}$$

Here n_q is the sequence length of the queries and n_k the sequence length of keys/values. We keep the same sequence-first convention used in Chapters 12 and 13: 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.

It is often helpful to name the attention-weight matrix

$$\mathbf{A} = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right),$$

where the softmax is applied *row-wise* (each query position gets a distribution over keys). Then the output is simply $\mathbf{Z} = \mathbf{A}\mathbf{V}$. The whole pipeline is differentiable, so the projection matrices that create Q, K, V can be learned by backpropagation.

Where do Q, K, V come from? Start from token vectors stacked into a matrix $\mathbf{X} \in \mathbb{R}^{n \times d_{\text{model}}}$ (one row per position). For a *single head*, learned projections produce

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^Q, \qquad \mathbf{K} = \mathbf{X}\mathbf{W}^K, \qquad \mathbf{V} = \mathbf{X}\mathbf{W}^V, \tag{14.6}$$

with $\mathbf{W}^Q, \mathbf{W}^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$ and $\mathbf{W}^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$. In self-attention, $\mathbf{X}$ is the same sequence for all three. In cross-attention (seq2seq), $\mathbf{Q}$ typically comes from decoder states while $\mathbf{K}, \mathbf{V}$ come from encoder states; the math is unchanged. Multi-head attention simply runs this projection-and-attend pattern several times in parallel with separate $\mathbf{W}_i^Q, \mathbf{W}_i^K, \mathbf{W}_i^V$.

Worked example: 2-token causal self-attention (one head)

Let $d_k = d_v = 2$ and $Q = K = V = \mathbf{I}_2$. Without masking, the scaled scores are

$$S = \frac{QK^\top}{\sqrt{d_k}} = \frac{1}{\sqrt{2}}\mathbf{I}_2.$$

A *causal mask* forbids looking into the future: for query index i, mask out all keys with index $j > i$ by setting those logits to $-\infty$ (so they vanish after the softmax).

Row 1 can only attend to itself, so $\alpha_{1\cdot} = [1, 0]$. Row 2 sees logits $[0,\ 1/\sqrt{2}]$, so

$$\begin{aligned} u &= e^{1/\sqrt{2}}, \\ \alpha_{2\cdot} &= \mathrm{softmax}\Big([0,\ 1/\sqrt{2}]\Big) = \left[\frac{1}{1+u},\ \frac{u}{1+u}\right] \\ &\approx [0.330238,\ 0.669762]. \end{aligned}$$

Because $V = \mathbf{I}_2$, the attention output equals the weight matrix: $\mathrm{Attn}(Q, K, V) = \alpha$.

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.

14.3 Self-attention vs. cross-attention

The same equations serve two distinct roles. *Self-attention* means Q, K, V come from the same sequence. This is the retrieval story turned inward: each position forms a query and pulls a weighted average over values from the whole sequence, producing a *contextualized* representation of that token. In decoder-only generation, self-attention is typically *causal*: the mask enforces that position t can only use positions $\leq t$.

This is why the same surface form can behave differently depending on what surrounds it. In the phrase "red flag," the useful information is often the idiom (a warning sign), not the literal color red or a physical flag. In "red coat," red is literal. Self-attention gives the model a mechanism to build token representations that reflect these different roles by mixing in different neighbors with different weights. Self-attention by itself does not know what "before" and "after" mean: without positional information the operation is permutation-equivariant. That is why we explicitly add positional encodings in Section 14.5.

Cross-attention is the seq2seq bridge. Here the queries come from the decoder states, but the keys and values come from the encoder outputs. In translation terms: each output position asks a question (query) and then pulls a weighted average over the source-side memory (keys/values) to decide what to emit next.

14.4 Multi-Head Attention (MHA)

One attention head gives you one similarity space. Multi-head attention gives you several in parallel: each head learns its own projections and can focus on different relations at the same time (local vs. global cues, syntactic vs. semantic signals, or different parts of an image-like grid). You should not read heads as guaranteed "modules" with fixed roles; the point is capacity and parallel views under the same weighted-average mechanism.

One intuition I find useful in language is that several relations matter at once. Some languages encode gender; some emphasize agreement; and word order can differ substantially across languages (subject–verb–object versus other patterns). During translation you may need to track several of these constraints simultaneously while still resolving local collocations. Multi-head attention gives the model multiple learned "views" at the same time: each head has its own projection matrices and can compute a different attention pattern over the same

sequence.

Multiple heads attend in parallel after learned linear projections:

$$\text{head}_i = \text{Attn}(\mathbf{X}\mathbf{W}_i^Q,\, \mathbf{X}\mathbf{W}_i^K,\, \mathbf{X}\mathbf{W}_i^V), \tag{14.7}$$

$$\text{MHA}(\mathbf{X}) = [\text{head}_1; \ldots; \text{head}_h]\, \mathbf{W}^O, \tag{14.8}$$

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 $\mathbf{W}^O \in \mathbb{R}^{(hd_v) \times d_{\text{model}}}$. For cross-attention, the formula is the same but keys/values come from encoder states: $\text{Attn}(\mathbf{X}_{\text{dec}}\mathbf{W}_i^Q, \mathbf{X}_{\text{enc}}\mathbf{W}_i^K, \mathbf{X}_{\text{enc}}\mathbf{W}_i^V)$. Figure 63 bundles scaled dot-product attention, multi-head concatenation, and the residual pre-LN block so the entire signal path is visible at a glance.

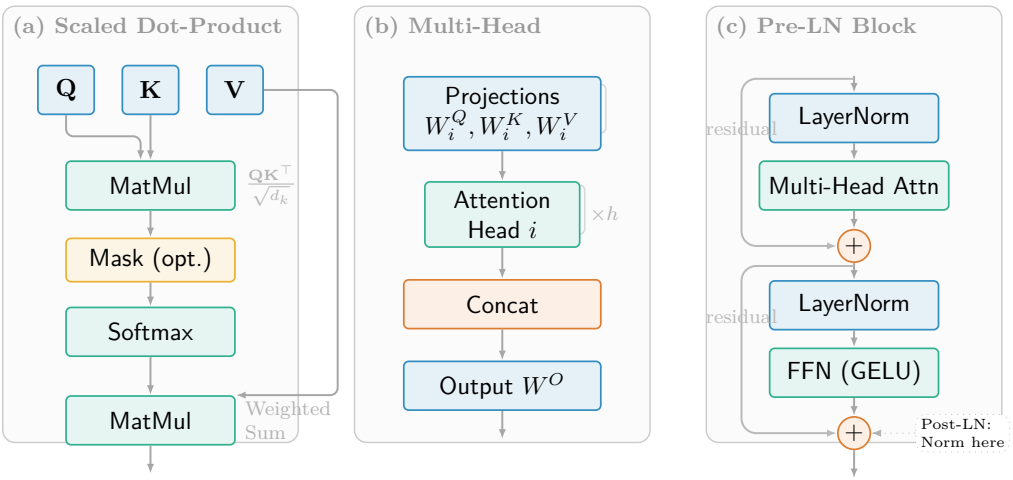

Figure 63: Reference schematic for the Transformer. Left: scaled dot-product attention. Center: multi-head concatenation with an output projection. Right: pre-LN encoder block combining attention, FFN, and residual connections; a post-LN variant simply moves each LayerNorm after its residual add (dotted alternative, not shown).

Figure 64 provides a quick visual summary of positional encoding, KV cache reuse, and LoRA adapters.

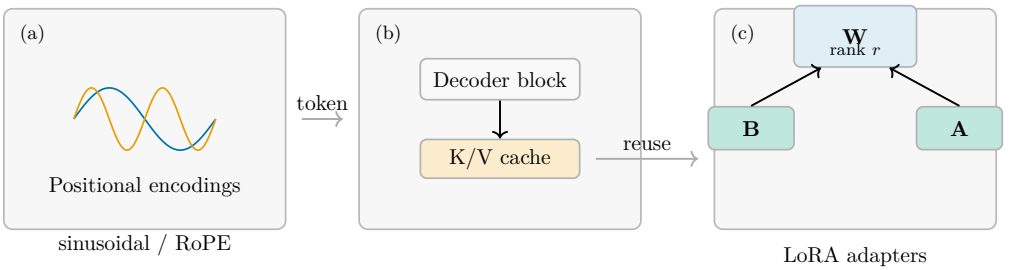

Figure 64: Transformer micro-views. Left: positional encodings (sinusoidal/rotary) add order information. Center: KV cache stores past keys/values so decoding a new token reuses prior context. Right: LoRA inserts low-rank adapters (B A) on top of a frozen weight matrix W for parameter-efficient tuning.

14.5 Positional Information

Transformers lack recurrence, so order has to be injected explicitly. A simple baseline (from Vaswani et al. (2017)) is a sinusoidal positional encoding: for position pos and feature index i,

$$\mathrm{PE}(\mathrm{pos},\, 2i) = \sin\!\left(\frac{\mathrm{pos}}{10000^{2i/d_{\mathrm{model}}}}\right), \tag{14.9}$$

$$\mathrm{PE}(\mathrm{pos},\, 2i+1) = \cos\!\left(\frac{\mathrm{pos}}{10000^{2i/d_{\mathrm{model}}}}\right). \tag{14.10}$$

You add this vector (or a learned alternative) to the token embedding at each position. The engineering point is not the specific constant; it is that different dimensions oscillate at different frequencies, so nearby positions look similar in some coordinates and far positions look different in others.

You can view the input to the first block as

$$\mathbf{H}_0 = \mathrm{Embed}(\mathrm{tokens}) + \mathrm{PosEnc}(\mathrm{positions}),$$

where both terms produce vectors in $\mathbb{R}^{d_{\mathrm{model}}}$ and the sum is taken position-wise.

In modern practice you will also see learned positional embeddings, relative-position schemes, and rotary position embeddings (RoPE). The chapter keeps sinusoidal PE as the clean reference, and we treat other variants as drop-in replacements that mainly change how well models extrapolate to longer contexts or shifting windows.

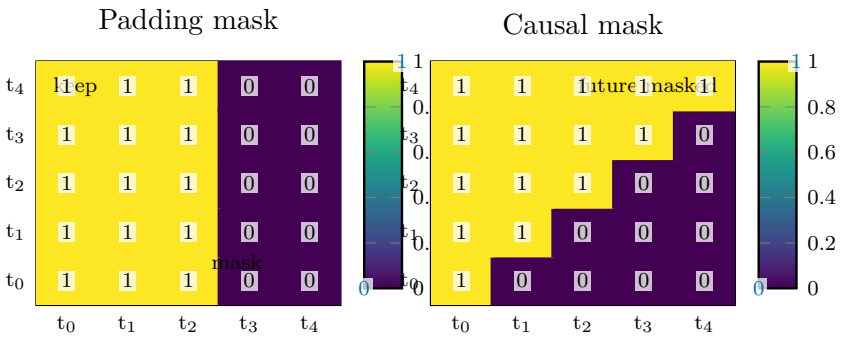

Figure 65: Attention masks as heatmaps (queries on rows, keys on columns). Left: padding mask zeroes out attention to padded positions of a shorter sequence in a packed batch. Right: causal mask enforces autoregressive flow by blocking attention to future tokens.

14.6 Objectives, masks, and model families

Two masks show up so often that it is worth naming them early. A *causal mask* forbids attention to future positions; this is what makes next-token generation well-defined. A *padding mask* forbids attention to padded positions inserted only to make batches rectangular. Both are easy to get wrong: a single missing mask can leak information from the future or silently change what the model is allowed to use. See Figure 65 for a concrete picture of both patterns (queries on rows, keys on columns).

One subtle point is *why* the causal mask is needed during training. At inference time the future does not exist, so a decoder cannot look ahead even if it wanted to. But during training we often feed the full target sequence (teacher forcing: provide the ground-truth previous tokens) so we can compute a loss at every position. Without a causal mask, the decoder would "cheat" by attending to future target tokens. The causal mask enforces the same information constraint at training time that you will have at decoding time.

The training objective is usually a cross-entropy (CE) loss, i.e., a negative log-likelihood (NLL) of the correct label under the model's predicted distribution. In sequence modeling, you typically sum (or average) that loss across time positions and across batch elements.

We will return to the main Transformer families (encoder-only vs. decoder-only vs. encoder–decoder) after we write down the block structure in Section 14.7.

The core idea is simple: families differ mostly by which masks they apply and which prediction problem they are trained on, not by a different attention mechanism.

14.7 Encoder/Decoder Stacks and Stabilizers

Transformer blocks are built from the same few ingredients repeated many times: attention, a small feed-forward network, and stability glue (residual connections, normalization, dropout). Residual connections keep a direct path for information and gradients: the input to a sublayer is added back to its output so the model can refine a representation without destroying it. LayerNorm stabilizes scale by normalizing each token vector across its feature dimension; unlike BatchNorm it does not depend on batch statistics, which matters for variable-length sequences and small batches.

In the *pre-LN* layout (common in modern implementations), each sublayer sees a normalized input and then its output is added back to the residual stream:

$$\mathbf{H}_1 = \mathbf{H} + \mathrm{MHA}(\mathrm{LayerNorm}(\mathbf{H})), \tag{14.11}$$

$$\mathbf{H}_{\text{out}} = \mathbf{H}_1 + \mathrm{FFN}(\mathrm{LayerNorm}(\mathbf{H}_1)). \tag{14.12}$$

The original Transformer applied LayerNorm after each residual add (post-LN); the attention and FFN computations are the same, only the placement of normalization changes.

The feed-forward sublayer (FFN) is applied independently to each position (row). In matrix form it is typically two linear layers with a nonlinearity:

$$\mathrm{FFN}(\mathbf{H}) = \sigma(\mathbf{H}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2, \tag{14.13}$$

where σ is commonly ReLU (original paper) or GELU (many modern stacks), $\mathbf{W}_1 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$, $\mathbf{W}_2 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$, and the biases are broadcast across positions. Dropout and label smoothing are common.

Implementation snapshot: block pseudocode, training defaults, and one step

Block pseudocode (pre-LN).

```
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, practical baseline).** AdamW with cosine decay and 1–3% warmup; LR $\sim 10^{-3}$ for small models, $1\text{–}2 \times 10^{-4}$ 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 and 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 (see Figure 64 and Section 14.14).

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 table $\mathbf{W}$ (written algebraically as a row-selection, e.g., $\mathbf{W}[w_t]$, in Chapter 13).

14.8 BERT vs. GPT vs. encoder–decoder

Once you have attention blocks, most of the "model family" distinctions come from two choices: (i) which directions a position is allowed to attend to, and (ii) what prediction problem you train on.

- **Encoder-only (BERT-style):** no causal mask; each token can use information from both left and right. Training commonly hides some input tokens and asks the model to predict the missing pieces (masked language modeling, MLM). A pooled vector (often a prepended `[CLS]` token) is then used as a sentence representation for classification.
- **Decoder-only (GPT-style):** causal mask; position t can only attend to positions $\leq t$. Training is next-token prediction (causal language modeling, CLM). Inference is the same loop run forward: predict the next token, append it, and repeat.
- **Encoder–decoder (seq2seq):** the encoder reads the source; the decoder generates the target with causal self-attention plus cross-attention into the encoder outputs. This is the cleanest fit for translation: each output position pulls a weighted average over source-side memory and then commits to one more token.

Some BERT variants also used "next sentence prediction" (NSP) as an auxiliary task historically; it is not essential for understanding the core encoder-only idea.

14.9 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 lin-

early in context length.

- **I/O-aware kernels** such as FlashAttention that stream tiles through SRAM so the $O(n^2)$ attention computation remains exact while memory stays manageable.

Relative/rotary position schemes and KV caching are summarized in Section 14.14 and Figure 64.

14.10 Fine-Tuning and Parameter-Efficient Adaptation

Pre-training gives you a general-purpose language model; fine-tuning adapts that model to a task, domain, or interaction style. *Full fine-tuning* updates all weights, which can yield the best performance when you have enough high-quality data and a stable objective, but it is also the easiest to overfit or to accidentally "forget" useful general behavior.

Parameter-efficient methods (LoRA, adapters, prefix/prompt tuning, and related variants) inject small trainable modules while freezing most of the base model, enabling rapid adaptation with lower memory and more predictable changes. Practically, PEFT is attractive when you want many task-specific variants of a shared base model, or when you need to keep the base weights fixed for deployment and auditing.

Audit hooks for adaptation. Regardless of whether you update all weights or only a small adapter, treat fine-tuning like any other ERM pipeline: keep a held-out evaluation set that matches the deployment slice, monitor calibration and failure modes (not just loss), and log the exact base checkpoint, tokenizer, and data snapshot so results are reproducible. When fine-tuning for instruction-following or conversational behavior, add explicit tests for regressions (refusals, hallucinations on factual probes, and brittleness to prompt variants) rather than relying on a single aggregate score.

14.11 Decoding and Evaluation

Training produces a distribution over the next token; decoding turns that distribution into an actual sequence. For decoder-only models, decoding runs autoregressively: predict $p_\theta(x_{t+1} \mid x_{1:t})$, choose a token, append it, and repeat. This is also where the KV cache matters (see Figure 64): you reuse past keys/values

so generating one more token does not require recomputing attention over the entire prefix from scratch.

Greedy decoding takes the argmax at each step; it is fast and often strong for short, factual completions, but it can get stuck in repetitive loops. Beam search keeps multiple partial hypotheses; it can improve likelihood but sometimes harms perceived quality in open-ended generation. Sampling (top-k, top-p nucleus) trades certainty for diversity; temperature controls how sharp or flat the distribution feels.

For evaluation, perplexity summarizes next-token performance for language modeling (see Chapter 13), but it does not tell you whether generations are useful, safe, or faithful. For downstream classification, prefer metrics that match the deployment slice (e.g., AUPRC for imbalanced problems) and keep decoding settings logged alongside checkpoints so results are reproducible.

14.12 Audit and failure modes (short list)

Audit and failure modes (engineering view)

- **Masking bugs:** missing/incorrect causal masks can leak future tokens; missing padding masks can let padding dominate attention in batched training.
- **Train/inference mismatch:** teacher forcing during training does not automatically tell you how errors compound during decoding; test the decoding strategy you plan to ship.
- **Long-context degradation:** attention enables long-range access, but quality can still decay with length; measure how performance changes as you increase context.
- **Calibration vs. correctness:** high probability is not a guarantee of correctness; audit reliability on slices and stress tests, not just average loss.
- **Reproducibility:** tokenizer choices, data filters, and decoding hyperparameters can swing results; log them as part of the experiment.

14.13 Alignment (Brief)

Post-training *alignment* shapes model behavior to match human preferences, safety constraints, and interaction norms. In broad terms, alignment objectives do not change the Transformer mechanics; they change what you reward during optimization (and therefore what behaviors are reinforced).

RLHF optimizes a policy against a learned reward model (with careful regularization to avoid drifting too far from the base model). Preference-based objectives such as DPO, KTO, or ORPO optimize directly from ranked pairs without a full reinforcement-learning loop.

Alignment is not a proof of correctness. Alignment can improve helpfulness and reduce obvious failures, but it can also introduce new ones (reward hacking, over-refusal, brittleness to prompt phrasing, or degraded performance off-distribution). Treat it as an engineering stage with explicit test suites and logging: evaluate on held-out tasks, check calibration and refusal behavior, and track changes relative to the pre-alignment model.

14.14 Advanced attention and efficiency notes (practitioner 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.
- **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.

Parameter-efficient tuning methods are covered in Section 14.10.

14.15 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)

Key takeaways

Terminology. Masked-LM and next-token LM are *self-supervised* (targets derived from input), not unsupervised.

- 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.

What to be able to do.

- Compute masked attention for a short sequence and explain why the mask enforces causality.
- Explain pre-LN vs. post-LN and why residual paths influence optimization stability.
- Describe KV caching and how it changes inference-time cost compared to training-time cost.

Common pitfalls to watch for.

- Incorrect masking (future leakage) or inconsistent tokenization between training and evaluation.
- Reporting speed or memory without stating context length, batch size, and caching assumptions.
- Treating decoding strategy as an afterthought; greedy, beam, and sampling regimes change observed quality.

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/s.
- 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.

If you are skipping ahead. After this chapter, the book pivots away from neural sequence models, so treat this chapter as the last stop for masking discipline and evaluation hygiene. If you need the embedding objectives and bias/deployment checklist, see Chapter 13.

Where we head next. Chapter 15 steps away from neural sequence models and re-enters the broader soft-computing toolkit (fuzzy logic and evolutionary ideas) previewed in Chapter 1. Read this chapter as the endpoint of the neural sequence thread: representation objectives from Chapter 13 plus masking/calibration discipline from Chapters 12 to 14.

Part II takeaways

- Representation and training are coupled: expressivity is only useful if gradients can reach the parameters.
- Depth and inductive bias trade parameters for structure (MLPs vs. CNNs vs. recurrence vs. attention).
- Stability tools recur across architectures: initialization, normalization, regularization, and validation-driven stopping.
- Sequence models turn memory into computation: unrolling, masking, and caching define what information can flow.

Part III: Soft computing and fuzzy reasoning

How to read Part III (rolling window)

- **A trilogy by design:** fuzzy sets → fuzzy relations → fuzzy inference; each chapter reuses the thermostat to add one layer of machinery.
- **Repetition is intentional:** concepts reappear as intuition, then as algebra, then as end-to-end behavior. Read repeated material as "preview vs. formalization vs. sanity-check."
- **Audit like an engineer:** keep universes/units explicit, pick operators explicitly, and verify edge cases numerically before tuning aesthetics.

Part III bridge: specification becomes the model

Parts I–II trained models by minimizing losses: we picked a hypothesis class, optimized it, and then audited generalization with careful validation. That approach is powerful when (i) you can collect enough data and (ii) the target behavior is well captured by a smooth objective.

Part III is about a different pain point: in many engineered systems the bottleneck is not data, but *specification*. The concepts you need are linguistic ("too warm", "slightly fast", "acceptable risk"), the boundaries are negotiable, and the decision logic often needs to remain auditable.

Fuzzy reasoning treats that interface explicitly: we write membership functions to represent concepts, pick operators to combine them, and assemble rules whose effects you can inspect. Everyday language is good at expressing graded categories; fuzzy logic turns that graded meaning into a mathematical object you can test, debug, and tune.

Read the repetition in this part as intentional: it is there to build familiarity with the machinery and to separate intuition from algebra. We revisit the same thermostat-style scenario as a preview, then as formal derivations, then as end-to-end behavior. Each pass adds one layer: fuzzy sets (building blocks), fuzzy relations (transfer and composition), and fuzzy inference (a complete controller).

When you get stuck, debug like an engineer: keep universes and units explicit, state your operator defaults, and test edge cases numerically before you tune aesthetics.

15 Introduction to Soft Computing

Parts I–II emphasized two complementary toolkits: probabilistic modeling and ERM (model–loss–optimize–audit), and biologically inspired learning (distributed representations trained by gradient descent). They shine when you can write down a smooth objective and collect enough data. Many engineered systems, however, are constrained less by data or compute than by specification: the relevant concepts are linguistic ("too warm," "slightly fast," "acceptable risk"), the boundaries are negotiable, and the decision logic must remain auditable.

Soft computing makes that vagueness explicit. Instead of insisting on exact equations everywhere, we represent human concepts with graded membership functions, combine them with transparent rules, and then tune those choices (and their trade-offs) with optimization rather than brittle hand tweaks. The shift is not away from rigor; it is toward placing approximations where they belong—at the interface between messy human predicates and systems that still have to act.

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.
- Work through a compact thermostat/autofocus controller example that grounds the design choices introduced throughout the fuzzy trilogy.

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.

Running example (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 returns throughout the fuzzy trilogy: first to parameterize membership functions (Chapter 16), then to transport labels across related universes (Chapter 17), and finally to assemble complete inference systems (Chapter 18).

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.

This is a probability story (uncertainty about which class is true). Fuzzy logic addresses a different interface: degrees of truth for linguistic predicates such as "warm" or "slightly fast," even when measurements are exact. Chapter 16 makes that distinction concrete by writing membership functions and checking them at specific inputs.

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 defined in Section 15.2; here we map its constituents to where they are developed in this book so you can follow the learning outcomes above without re-reading the same motivation.

- **Fuzzy logic:** this chapter and the fuzzy trilogy Chapters 16 to 18 build membership functions, rules, and inference pipelines.
- **Neurocomputing and hybrids:** neural models live in Chapters 3 to 14; neuro-fuzzy hybrids such as ANFIS (Jang, 1993) sit at the interface.
- **Probabilistic modeling:** the statistical strand in Chapters 3 to 4 remains complementary to fuzzy possibility views (Dubois and Prade, 1988).
- **Evolutionary computation:** Chapter 19 shows how evolutionary search tunes rule bases and membership parameters (Herrera and Lozano, 2008; Ishibuchi and Nakashima, 2007).

Table 7 maps crisp logic operators to their fuzzy counterparts.

Table 6: Fuzzy vs. probabilistic reasoning. Distinguishes randomness (probability) from graded concepts (fuzziness) when interpreting uncertainty.

	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: Boolean vs. fuzzy operators. Mapping from crisp logic rules to graded operators for fuzzy inference.

	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 $1 - a$

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.

Fuzzy logic as a mathematical language. Zadeh's insight was to synthesize these classical mathematical languages into a framework with graded truth values; the next section builds the intuition and contrasts crisp versus fuzzy reasoning.

15.13 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.

15.14 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 16.8) 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`; `Falling`, `Stable`, `Rising`); 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 `Rising` 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; Chapter 17 provides the transfer operators (extension principle, projection, composition) that move fuzzy labels between related variables; and Chapter 18 assembles full inference and defuzzification. Chapter 19 later returns to the other soft-computing pillar: evolutionary search as a tuning tool for rule bases and memberships.

15.15 Wrapping up soft computing and previewing the fuzzy trilogy

This chapter set the motivation: when specifications are linguistic and boundaries are negotiable, it is often better to model that vagueness explicitly than to force brittle crisp thresholds. The next three chapters turn that motivation into a workflow you can execute and audit.

Fuzzy Sets Recap See Section 15.14 for the membership-function definition and Section 15.13 for the crisp-versus-fuzzy intuition; here we keep a final example and the forward pointer.

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, transformations, and inference Chapters 16 to 18 pick up the thermostat running example end to end. Chapter 16 constructs membership functions and operators (the primitives). Chapter 17 transports fuzzy labels across related universes via the extension principle (the transfer layer). Chapter 18 then assembles complete Mamdani/Sugeno inference and defuzzification (the decision layer). Keep this overview handy: it is the conceptual map, while the fuzzy trilogy works through the algebra and the numeric sanity checks.

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.

Minimum viable mastery.

- Define what is being approximated (truth values, search steps, or function classes) in each soft-computing pillar.
- Explain why operator choices matter (they encode semantics and shape the resulting decision surfaces).
- Connect fuzzy-set primitives to the later inference pipeline (fuzzify, combine, imply, aggregate, defuzzify).

Common pitfalls.

- Mixing operator families inconsistently across a pipeline and then debugging symptoms at the end.
- Treating "soft" as a license to skip validation: soft methods still require measurable objectives and checks.
- Ignoring scaling and units when defining universes and membership functions.

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.

If you are skipping ahead. When you reach Chapters 16 to 18, keep the operator choices explicit and consistent. Many formatting and interpretation problems later come from hidden operator defaults.

Where we head next. Chapters 16 to 18 carry the thermostat running example end-to-end: Chapter 16 builds membership functions, Chapter 17 transports information across universes through relations, and Chapter 18 assembles full inference plus defuzzification.

16 Fuzzy Sets and Membership Functions: Foundations and Representations

Building on Chapter 15, we formalize the foundations of fuzzy logic: universes of discourse, fuzzy sets, membership functions, and the operators used throughout the fuzzy trilogy. You will see some ideas from Chapter 15 again, but now in a form you can compute and sanity-check numerically (a deliberate preview→formalization pattern). These tools define the thermostat labels used later and prepare the transfer/inference steps developed in Chapters 17 to 18.

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.

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.

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.

16.1 Motivating example: designing a membership function from measurements

Consider the thermostat's temperature error $e = T_{\text{room}} - T_{\text{set}}$ in degrees Celsius (so $e < 0$ means "too cold"). We want a linguistic label *Cold* that is clearly true when the room is far below setpoint, clearly false when the room is at/above setpoint, and graded in between. A simple, auditable choice is a piecewise-linear

"shoulder" membership:

$$\mu_{\text{Cold}}(e) = \begin{cases} 1, & e \le -4, \\ \frac{0-e}{4}, & -4 < e < 0, \\ 0, & e \ge 0. \end{cases} \tag{16.1}$$

With this design, $e = -2$ yields $\mu_{\text{Cold}}(-2) = 0.5$: "cold" is half true. Later chapters reuse this shoulder *shape* but may shift its breakpoints to reflect different sensors, comfort bands, or operating assumptions. This number is not a probability; it is a degree of truth for a linguistic predicate. The rest of this chapter is about making these mappings explicit (shapes, overlaps, and operators) so they can be inspected and tuned rather than assumed.

16.2 Fuzzy sets and the universe of discourse

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).

16.3 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 $x \in X$,

$$\mu_A(x) \in [0, 1]. \tag{16.2}$$

The fuzzy set A can be represented as the collection of ordered pairs:

$$A = \{(x, \mu_A(x)) \mid x \in X\}. \tag{16.3}$$

16.4 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.

16.4.1 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:

$$A = \{(x_1, \mu_A(x_1)), (x_2, \mu_A(x_2)), \ldots, (x_n, \mu_A(x_n))\}. \tag{16.4}$$

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,

$$A = \{(2, 0.1), (3, 0.3), (4, 0.7)\}.$$

16.4.2 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:

$$A = \int_{x \in X} \mu_A(x)/x, \tag{16.5}$$

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:

$$\mu_A(x) = \max\left(0, 1 - \frac{|x - c|}{w}\right). \tag{16.6}$$

This function assigns membership 1 at $x = c$, decreasing linearly to zero at $x = c \pm w$.

16.5 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.

16.6 Membership Functions in Fuzzy Sets

With the universe X fixed and the concept A named, the membership function $\mu_A : X \to [0, 1]$ 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

$$\mu_A(x) = \begin{cases} 0, & x \leq a \\ \frac{x-a}{b-a}, & a < x \leq b \\ \frac{c-x}{c-b}, & b < x < c \\ 0, & x \geq c \end{cases} \tag{16.7}$$

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 $a < b \leq c < d$:

$$\mu_A(x) = \begin{cases} 0, & x \leq a \\ \frac{x-a}{b-a}, & a < x \leq b \\ 1, & b < x \leq c \\ \frac{d-x}{d-c}, & c < x < d \\ 0, & x \geq d \end{cases} \tag{16.8}$$

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):

$$\mu_A(x) = \exp\left(-\frac{(x-c)^2}{2\sigma^2}\right). \tag{16.9}$$

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:

$$\mu_A(x) = \frac{1}{1 + \left|\frac{x-c}{a}\right|^{2b}}. \tag{16.10}$$

This function allows control over the width and slope of the membership curve, interpolating between shapes similar to triangular and Gaussian functions.

16.7 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

$$\mu_B(x) = \begin{cases} 0, & x \le 75, \\ \dfrac{x-75}{5}, & 75 < x \le 80, \\ 1, & 80 < x \le 85, \\ \dfrac{90-x}{5}, & 85 < x \le 90, \\ 0, & x > 90, \end{cases}$$

so a borderline score such as $x = 79$ yields $\mu_C(79) = (80 - 79)/5 = 0.2$ and $\mu_B(79) = (79 - 75$ th analogous expressions for the zoids capture the intuition that a elong to both C and B to differen

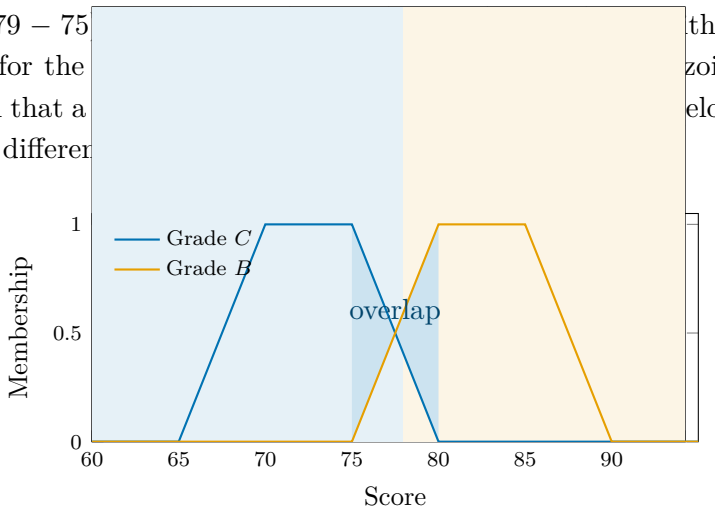

Figure 66: Trapezoidal membership functions for grades C and B with the overlapping region shaded. Scores near 78–82 partially satisfy both grade definitions. Use it when designing overlapping grade/linguistic bins so boundary cases behave smoothly.

Figure 67 is the overlap diagnostic used when tuning membership coverage.

16.8 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`. For example, at $x = 22$ we have $\mu_{\text{Medium}}(22) = (25 - 22)/5 = 0.6$ and $\mu_{\text{Large}}(22) = (22 - 20)/5 = 0.4$.

Figure 68 compares operator shapes when choosing conjunction behavior in rule aggregation.

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)
```

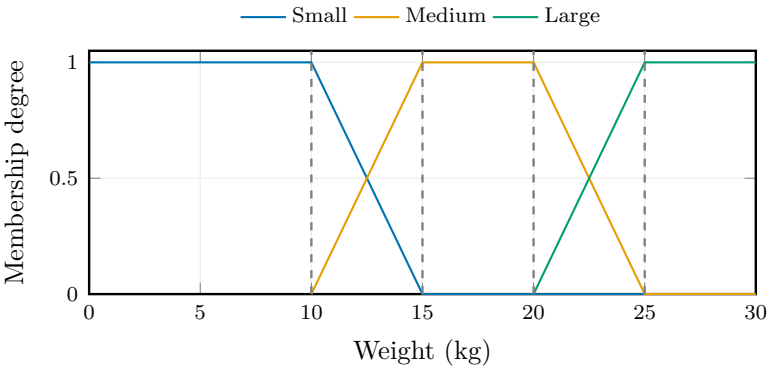

Figure 67: Overlapping membership functions for the "Small", "Medium", and "Large" weight labels. The shaded overlaps capture gradual transitions. Use it when tuning membership overlaps so small numeric changes do not cause abrupt rule changes.

```
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")
plt.plot(x, mu_med, label="Medium")
plt.plot(x, mu_large, label="Large")
plt.legend(); plt.show()
```

16.9 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:

$$\text{supp}(A) = \{x \in X \mid \mu_A(x) > 0\}. \tag{16.11}$$

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:

$$\text{core}(A) = \{x \in X \mid \mu_A(x) = 1\}. \tag{16.12}$$

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:

$$\mu_A(c_A^-) = \mu_A(c_A^+) = 0.5. \tag{16.13}$$

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.

16.10 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 frequency. 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).

16.11 Fuzzy Set Operations

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.

Notation handoff

In the fuzzy trilogy, $\mu_A(x)$ always denotes membership grade, T denotes a t-norm when generalized conjunction is needed, and S denotes an s-norm for generalized union. If these symbols clash with other parts, use the local chapter meaning and consult Appendix D.

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:

$$\mu_{A \cup B}(x) = \max\big(\mu_A(x), \mu_B(x)\big). \tag{16.14}$$

This generalizes the classical union by taking the maximum membership degree at each element.

Intersection The intersection $A \cap B$ is defined by:

$$\mu_{A \cap B}(x) = \min\big(\mu_A(x), \mu_B(x)\big). \tag{16.15}$$

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:

$$\mu_{A^c}(x) = 1 - \mu_A(x). \tag{16.16}$$

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:

$$\mu_{(A\cap B)^c}(x) = \mu_{A^c\cup B^c}(x) = \max\big(1-\mu_A(x),\, 1-\mu_B(x)\big), \tag{16.17}$$

$$\mu_{(A\cup B)^c}(x) = \mu_{A^c\cap B^c}(x) = \min\big(1-\mu_A(x),\, 1-\mu_B(x)\big). \tag{16.18}$$

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 (16.14) to (16.16) 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.

Graphical interpretation For continuous universes, the union and intersection membership functions are the pointwise maximum and minimum of the two curves, respectively. A complement mirrors memberships around $\mu = 0.5$: every membership degree m maps to $1-m$. When you sketch these operations quickly, plot both curves on the same axes and take the upper (union) or lower (intersection) envelope. The same pointwise definitions apply to discrete universes; you simply compute the max/min values at each listed x_i.

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:

$$\mu_{A\cdot B}(x) = \mu_A(x) \cdot \mu_B(x), \quad \forall x \in X. \tag{16.19}$$

Scalar Multiplication Given a scalar $\alpha \in [0, 1]$, scalar multiplication of a fuzzy set A is:

$$\mu_{\alpha A}(x) = \alpha \cdot \mu_A(x), \quad \forall x \in X. \tag{16.20}$$

Algebraic Sum The algebraic sum of fuzzy sets A and B is given by:

$$\mu_{A+B}(x) = \mu_A(x) + \mu_B(x) - \mu_A(x) \cdot \mu_B(x), \quad \forall x \in X. \tag{16.21}$$

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:

$$\mu_{A-B}(x) = \mu_A(x) \wedge \big(1 - \mu_B(x)\big) = \min\big(\mu_A(x), 1 - \mu_B(x)\big), \tag{16.22}$$

where $\wedge$ denotes the minimum operator.

Bounded Difference An alternative definition of difference is the bounded difference:

$$\mu_{A\ominus B}(x) = \max\big(0, \mu_A(x) - \mu_B(x)\big). \tag{16.23}$$

Remarks:

- The difference operation in (16.22) corresponds to the intersection of A with the complement of B.
- The bounded difference in (16.23) 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

Use the pointwise definitions in (16.14)–(16.15) to compute unions or intersections for any pair of fuzzy sets; the next subsection lifts these operations to relations via Cartesian products.

Example. Let $X = \{x_1, x_2, x_3\}$ and define memberships in the order (x_1, x_2, x_3):

$$\mu_A = \{0.2,\ 0.7,\ 0.4\}, \qquad \mu_B = \{0.6,\ 0.3,\ 0.9\}.$$

Then

$$\mu_{A\cup B} = \{\max(0.2, 0.6),\ \max(0.7, 0.3),\ \max(0.4, 0.9)\} = \{0.6,\ 0.7,\ 0.9\},$$
$$\mu_{A\cap B} = \{\min(0.2, 0.6),\ \min(0.7, 0.3),\ \min(0.4, 0.9)\} = \{0.2,\ 0.3,\ 0.4\}.$$

The union keeps the larger membership at each point, while the intersection keeps the smaller.

16.14 Cartesian Product of Fuzzy Sets

Using the membership-function definition from Section 16.9, the *Cartesian product* of two fuzzy sets A on X and B on Y 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

$$\mu_{A\times B}(x, y) = \min\big(\mu_A(x), \mu_B(y)\big), \quad \forall x \in X, y \in Y. \tag{16.24}$$

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:

$$\mu_{A\cap B}(x) = \mu_{B\cap A}(x), \tag{16.25}$$

$$\mu_{A\cup B}(x) = \mu_{B\cup A}(x). \tag{16.26}$$

Associativity:

$$\mu_{(A\cap B)\cap C}(x) = \mu_{A\cap(B\cap C)}(x), \tag{16.27}$$

$$\mu_{(A\cup B)\cup C}(x) = \mu_{A\cup(B\cup C)}(x). \tag{16.28}$$

Distributivity:

$$\mu_{A\cup(B\cap C)}(x) = \mu_{(A\cup B)\cap(A\cup C)}(x), \tag{16.29}$$

$$\mu_{A\cap(B\cup C)}(x) = \mu_{(A\cap B)\cup(A\cap C)}(x). \tag{16.30}$$

Identity Elements:

$$\mu_{A\cup\emptyset}(x) = \mu_A(x), \tag{16.31}$$

$$\mu_{A\cap X}(x) = \mu_A(x), \tag{16.32}$$

where $\emptyset$ is the empty fuzzy set with membership zero everywhere, and X is the universal fuzzy set with membership one everywhere.

Involution:

$$\mu_{(A^c)^c}(x) = \mu_A(x), \tag{16.33}$$

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 (16.14) to (16.16), 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 (16.16). 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

$$C_p(\mu) = \frac{1-\mu}{1+p\mu}, \qquad p \geq 0, \tag{16.34}$$

which reduces to the standard complement when $p = 0$.

Another simple family is a power-law negation:

$$C_p(\mu) = (1-\mu)^p, \qquad p > 0, \tag{16.35}$$

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

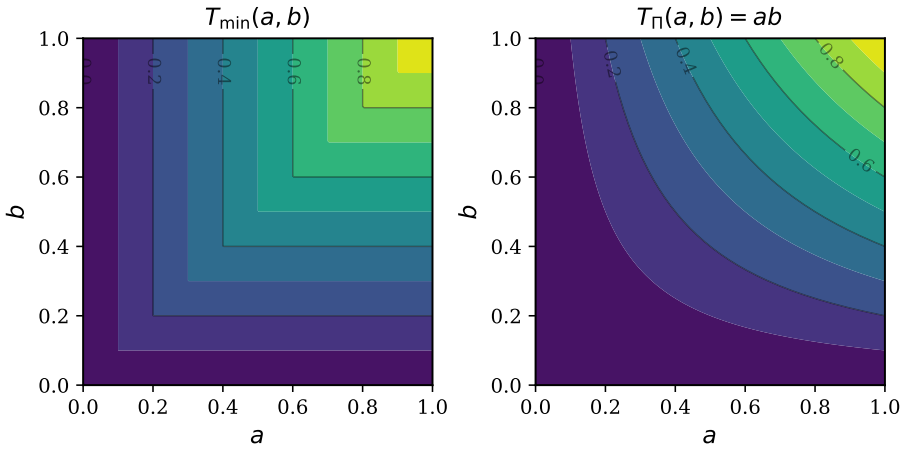

Figure 68: Fuzzy AND surfaces comparing minimum versus product t-norms; analogous OR surfaces show similar differences. Choices here influence rule aggregation in Chapter 18. Use it when deciding whether conjunction should behave like a conservative minimum or a multiplicative attenuation.

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 $C(1) = 0$.
- **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

We use the t-norm and s-norm definitions from the previous two subsections; here we focus on their complementarity via negation.

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:

$$T(\mu_A(x), \mu_B(x)) = 1 - S(1 - \mu_A(x), 1 - \mu_B(x)), \tag{16.36}$$

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 (16.36).

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 $\mathrm{Inc}(A,B)$, measures how much A is a subset of B.

One way to define this grade is:

$$\mathrm{Inc}(A,B) = \inf_{x \in X} I\big(\mu_A(x), \mu_B(x)\big), \tag{16.37}$$

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

$$\mathrm{Inc}(A,B) = \inf_{x \in X} T\big(\mu_A(x), \mu_B(x)\big),$$

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:

$$\mathrm{Eq}(A, B) = \inf_{x \in X} J\big(\mu_A(x), \mu_B(x)\big), \tag{16.38}$$

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:

$$\text{Dilation:} \quad \mu_{A^{(d)}}(x) = \big(\mu_A(x)\big)^{1/\alpha}, \quad \alpha \geq 1, \tag{16.39}$$

$$\text{Contraction:} \quad \mu_{A^{(c)}}(x) = \big(\mu_A(x)\big)^{\beta}, \quad \beta \geq 1. \tag{16.40}$$

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

This chapter completes the toolkit for generating new membership functions from existing ones via fuzzy-set operations. Membership functions encode graded belonging over a universe of discourse, and algebraic manipulation of those functions is central to fuzzy logic and fuzzy inference.

Two transfer cues matter for what comes next. First, treat membership functions as *objects you can manipulate*: once the shapes are written down, you can reason about overlaps, negations, and edge cases without guessing. Second,

the same max/min mindset will reappear when we move from sets to *relations*: we will build fuzzy relations over Cartesian products and then compose them (a matrix-like operation) to propagate graded information between universes.

As you generate new membership functions below, keep a short audit checklist: verify the output stays in $[0, 1]$, check endpoints and monotonicity where intended, and confirm that the universe/units match the linguistic meaning. Those checks are invariants you will want to preserve later when you tune breakpoints, operator defaults, or rule bases.

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:

$$\text{Age} \quad (x), \quad \text{Exercise Level} \quad (e)$$

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:

$$\begin{aligned} &\text{IF Age is old AND Exercise is high} \\ &\qquad\qquad \text{THEN Health is good} \end{aligned}$$

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.

Table 8 serves as the operator-choice checklist in the Mamdani design discussion.

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

Table 8: 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. Use it when choosing default operators for a Mamdani-style pipeline and predicting qualitative behavior.

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.

- 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 $T = 27\,℃$,

$$\begin{aligned}\mu_{\text{Cold}}(27) &= 0,\\ \mu_{\text{Warm}}(27) &= \frac{30-27}{10} = 0.3,\\ \mu_{\text{Hot}}(27) &= \frac{27-25}{15} = \frac{2}{15} \approx 0.133.\end{aligned}$$

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 69 for the membership functions and clipping levels used in this example. Practical tip: libraries such as `scikit-fuzzy` provide a tested `defuzz`

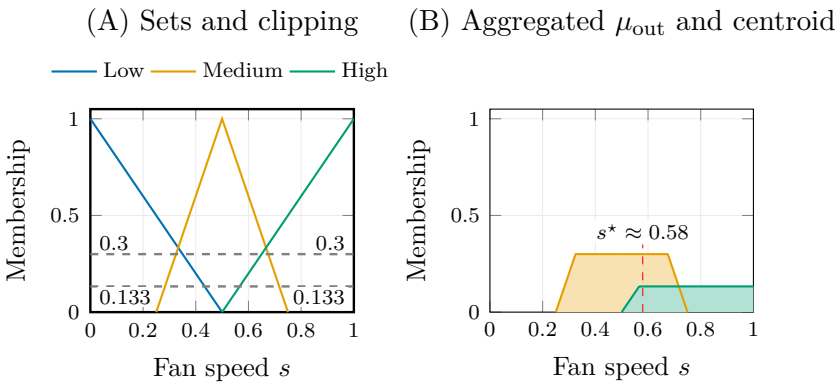

Figure 69: End-to-end fuzzy inference example. (A) Consequent membership functions with clipping levels from firing strengths at T = 27 deg C. (B) Aggregated output set (max of truncated consequents) and a centroid marker near s* approx 0.58. Use it when sanity-checking clipping, aggregation, and centroid defuzzification end to end.

(centroid) routine; when in doubt, compute the centroid numerically rather than relying on heuristic convex combinations.

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.

Minimum viable mastery.

- Construct a universe of discourse, define overlapping memberships, and compute degrees for concrete inputs.
- Apply basic operations (min/max, product, complements) and predict how the choice changes shape.
- Translate a design intent ("smooth", "conservative", "aggressive") into membership overlap and operator choices.

Common pitfalls.

- Setting memberships without checking units/scales, producing labels that never activate or always saturate.
- Using too many overlapping labels without justification (hard to tune, hard to interpret).
- Tuning by aesthetics alone instead of checking downstream sensitivity and stability.

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.

If you are skipping ahead. The rest of the fuzzy chapters build on these primitives. If you find later rule outputs unstable, the first place to revisit is the universe scaling and overlap choices here.

Where we head next. Chapter 17 moves from set-level primitives to *transfer*: how to push fuzzy labels through mappings between related universes (extension principle, projection, and composition). Chapter 18 then uses those relation tools to assemble complete rule composition and defuzzification pipelines.

17 Fuzzy Set Transformations Between Related Universes

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 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. This chapter develops that transfer layer so Chapter 18 can assemble full inference pipelines.

Learning Outcomes

- Apply the extension principle (single and multi-variable) to transport fuzzy knowledge across domains.
- Select appropriate t-norms/t-conorms 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.

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).

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.

Rolling-window note (what changes vs. Chapter 16). In Chapter 16, the main design work was *within* a universe: choose membership shapes, overlaps, and operators. Here the memberships are assumed given; the new question is *between* universes: how does a fuzzy label move through a mapping without losing meaning? Treat this chapter as the transfer layer that makes the later inference pipeline (Chapter 18) reusable across sensors, units, and derived variables.

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.

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.

Notation. When we introduce a general t-norm T, it appears explicitly (e.g., in (17.2) and (17.8)). For symbol overloads when reading across parts, see Appendix D.

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 preimage 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 (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 preimage sets $\{x \mid f(x) = y\}$ are well-defined and the supremum exists.

17.5 Interpretation

For single-input mappings, the output membership is the strongest preimage; when multiple fuzzy inputs are involved we first combine their memberships with a chosen t-norm (cf. (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 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 $y = f(x)$. The extension principle itself was defined in (17.1); here we work through a concrete discrete example that instantiates that definition.

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:

$$Y = \{0^2, (-1)^2, 1^2, 2^2\} = \{0, 1, 4\}.$$

Calculating explicitly:

$$\mu_B(0) = \mu_A(0) = 0.141,$$
$$\mu_B(1) = \max\{\mu_A(-1), \mu_A(1)\} = \max\{0.340, 0.242\} = 0.340,$$
$$\mu_B(4) = \mu_A(2) = 0.4.$$

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 preimages.

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 70 walks through a simple mapping $y = x^2$, showing how memberships on X lift to memberships on Y via the supremum over all preimages that map to the same point.

Figure 71 makes the non-monotone alpha-cut mapping explicit for piecewise image transforms.

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 $(-1, 0, 1, 2)$:

$$\mu_{A_1} = \{0.4, 0.7, 0.5, 0.13\}, \quad \mu_{A_2} = \{0.5, 0.1, 0.4, 0.7\}.$$

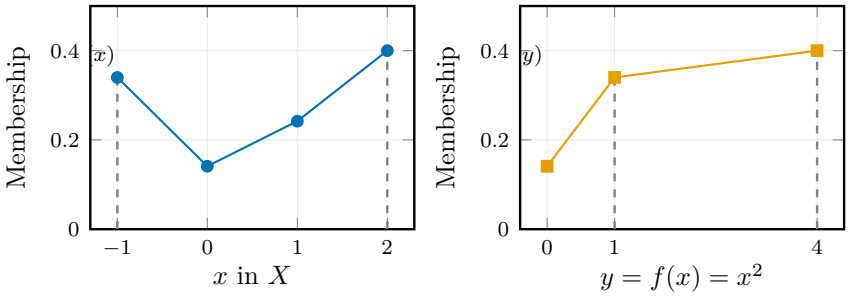

Figure 70: Mapping a fuzzy set through the function "y = x-squared". The membership at an output value y is the supremum over all preimages x that map to y; shared images such as $x = \pm 1$ map to y = 1 using the maximum membership. Helpful when applying the extension principle to a non-invertible function.

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 For two variables, the extension principle becomes:

$$\mu_B(y) = \sup_{(x_1,x_2):f(x_1,x_2)=y} \min\{\mu_{A_1}(x_1), \mu_{A_2}(x_2)\}. \tag{17.2}$$

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.

The numeric workflow is consistent: list all preimages (x_1, x_2) with $f(x_1, x_2) = y$, compute the minimum membership for each pair, then take the maximum.

Example: Compute $\mu_B(0)$.

The only pair with $x_1^2 + x_2^2 = 0$ is $(0, 0)$, so

$$\mu_B(0) = \min\{\mu_{A_1}(0), \mu_{A_2}(0)\} = \min\{0.7, 0.1\} = 0.1.$$

Example: Compute $\mu_B(1)$.
The pairs (x_1, x_2) such that $x_1^2 + x_2^2 = 1$ are:

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

The minimum membership values for each pair are

$$\begin{aligned}
\min\{\mu_{A_1}(-1), \mu_{A_2}(0)\} &= \min\{0.4, 0.1\} = 0.1, \\
\min\{\mu_{A_1}(0), \mu_{A_2}(-1)\} &= \min\{0.7, 0.5\} = 0.5, \\
\min\{\mu_{A_1}(1), \mu_{A_2}(0)\} &= \min\{0.5, 0.1\} = 0.1, \\
\min\{\mu_{A_1}(0), \mu_{A_2}(1)\} &= \min\{0.7, 0.4\} = 0.4.
\end{aligned}$$

Taking the supremum gives $\mu_B(1) = \max\{0.1, 0.5, 0.1, 0.4\} = 0.5$.

17.7 Extension Principle Recap and Projection Operations

We now focus on computational choices (discretization vs. alpha-cuts) before moving on to projection operations.

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.

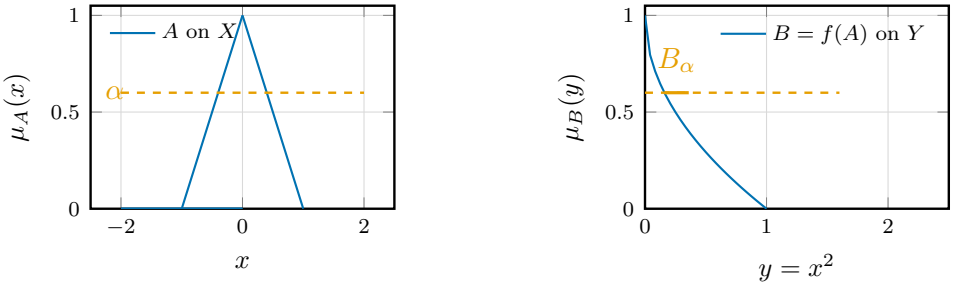

Figure 71: Alpha-cuts under the non-monotone map "y = x-squared". A symmetric triangular fuzzy set on X maps to a right-skewed fuzzy set on Y. Each alpha-cut on A splits into two intervals whose images union to the output alpha-cut. Use this to propagate a fuzzy set through a non-monotone map via alpha-cuts.

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.

Figure 72 visualizes how relation entries project into the corresponding marginals.

17.8 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 $X \times Y$. We define projection operators in this section and later reuse them in the dimensional extension discussion; the goal here is to fix notation and walk a concrete table

example.

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

$$\mu_R(x, y) = T(\mu_A(x), \mu_B(y)), \tag{17.3}$$

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: Popular t-norms and their typical roles. Reference when choosing a default conjunction operator and understanding its qualitative behavior.

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.

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

$$\mu_{\pi_X(R)}(x) = \sup_{y \in Y} \mu_R(x, y). \tag{17.4}$$

Definition (Projection onto Y). Similarly, the projection of R onto Y, denoted $\pi_Y(R)$, is defined by

$$\mu_{\pi_Y(R)}(y) = \sup_{x \in X} \mu_R(x, y). \tag{17.5}$$

Total Projection The *total projection* of R is the maximum membership value over the entire relation:

$$\mu_{\pi_{\text{total}}(R)} = \sup_{x \in X, y \in Y} \mu_R(x, y). \tag{17.6}$$

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.

	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 72: 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. Use it to build rule antecedents from a relation and verify which universe each projection inhabits.

Example (continued) Using the previous example matrix for μ_R:

$$\mu_R = \begin{bmatrix} 0.5 & 0.5 \\ 0.8 & 0.9 \end{bmatrix},$$

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

$$\mu_{\pi_{\text{total}}(R)} = \max\{0.5, 0.8, 0.5, 0.9\} = 0.9.$$

17.9 Dimensional Extension and Projection in Fuzzy Set Operations

In practice, we frequently need to combine fuzzy sets and fuzzy relations that live on *different* universes of discourse. For example, one object may be defined on a one-dimensional universe X, while another lives on a product space $X \times Y$ (as relations do). Before we can take unions/intersections or compose rules, we must reconcile dimensions. Two operators do the heavy lifting: *cylindrical extension* lifts a set into a higher-dimensional space without changing its meaning, and *projection* collapses a relation back onto the universe we want to summarize.

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:

$$\mu_{A^*}(x, y) = \mu_A(x), \quad \forall x \in X, y \in Y. \tag{17.7}$$

This operation "copies" the membership values of A uniformly along the Y-dimension, resulting in a fuzzy set over $X \times Y$.

Projection Projection collapses relations back onto a target universe; use the definitions in (17.4)–(17.5) and apply them after cylindrical extension to recover the desired marginal.

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, i.e.,

$$\mu_{A^*} = \begin{bmatrix} 0.5 & 0.5 & 0.5 \\ 0.7 & 0.7 & 0.7 \end{bmatrix},$$

with rows corresponding to x_1, x_2 and columns to y_1, y_2, y_3. 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.

17.10 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.

We seek the composed relation $R \subseteq X \times Z$ that relates X directly to Z by composing R_1 and R_2.

Composition of Fuzzy Relations The composition $R = R_1 \circ R_2$ is defined by:

$$\mu_R(x, z) = \sup_{y \in Y} T\big(\mu_{R_1}(x, y), \mu_{R_2}(y, z)\big). \tag{17.8}$$

We default to the *sup–min* (max–min) composition by taking $T(a, b) = \min(a, b)$; other t-norms are listed in 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 t-conorm to instantiate other composition families.

17.11 Properties of Fuzzy Relation Composition

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 closing section is to wrap up fuzzy relation composition, connect max–min to its general sup–T form, and record the algebraic properties that will later matter when relations are used as building blocks for fuzzy inference systems.

Generalization: sup–T composition Equation (17.8) already defines the general sup–T composition. A valid t-norm $T : [0,1]^2 \to [0,1]$ is commutative, associative, monotone in each argument, and satisfies $T(a,1) = a$; common choices include the minimum, product, and Łukasiewicz operators. Setting $T(a,b) = \min(a,b)$ recovers the standard max–min composition used throughout this chapter.

Example pointer The max–min example in Section 17.10 already walks through a concrete composition; with that numeric pattern in mind, we now focus on the properties that hold for any valid sup–T operator.

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.

17.12 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.

Minimum viable mastery.

- Given $y = f(x)$, compute $\mu_Y(y)$ via inverse mappings and a chosen aggregation rule.
- State when the mapping is one-to-one vs. many-to-one and how that changes the computation.
- Track domain restrictions explicitly so transformed sets respect feasibility (e.g., square-root domains).

Common pitfalls.

- Dropping multi-preimage cases and silently producing overconfident outputs.
- Hiding domain restrictions, leading to membership mass on invalid regions.
- Mixing notations for T, sup, and complements across examples (hard to audit later).

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.

If you are skipping ahead. The extension principle is the bookkeeping layer for the full inference pipeline. When you reach Chapter 18, every "rule output" is an instance of the same transfer-and-aggregate pattern.

Where we head next. Chapter 18 operationalizes these relation tools: each rule induces a fuzzy relation on input-output space, then sup-T composition and projection produce the implied output set before aggregation and defuzzification. The same defaults (max–min with standard complement) carry over, as summarized in (18.2)–(18.6).

18 Fuzzy Inference Systems: Rule Composition and Output Calculation

Building on Chapter 17, where transfer operators (projection, composition, and the extension principle) move fuzzy information between related universes, we now assemble complete fuzzy inference systems (FIS): rule composition and output calculation. This closes the fuzzy trilogy by turning set/relation machinery into end-to-end decisions.

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.

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.

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 Stable THEN heater power is High") in mind; the formulas below map directly onto that setup.

18.1 Context and Motivation

Notation handoff

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. If symbols conflict with earlier chapters, use local definitions and refer to Appendix D.

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:

$$\mu_{B_{\text{agg}}}(y) = \max_j \mu_{B'_j}(y). \tag{18.6}$$

We denote this aggregated set by $\mu_{B_{\text{agg}}}$; in the worked thermostat example we write $\mu_{\text{out}}(u)$ for the same quantity to emphasize the output variable.

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

For a compact algorithmic view, see the pipeline box after the worked example; it instantiates these same steps with explicit operators.

Risk & audit

- **Rule explosion:** too many overlapping rules can make behavior opaque; prune with coverage and conflict diagnostics.
- **Operator mismatch:** min/product implication and max/sum aggregation produce different surfaces; choose once and document why.
- **Membership miscalibration:** poorly scaled sets can overfire or underfire rules; audit control-surface smoothness and boundary behavior.
- **Defuzzification sensitivity:** centroid vs. weighted average can shift outputs materially; test with edge-case inputs.
- **Deployment drift:** sensor scaling changes break rule semantics; version normalization and monitor post-deployment residuals.

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.

18.6 Worked example: thermostat inference with numbers

We now run one complete inference pass for a compact thermostat rule base with two inputs and one output. The goal is not to defend a particular membership design, but to show how the algebra produces a concrete output that you can compute, audit, and debug.

Universes and memberships. Let temperature error $e \in [-5, 5]\,^\circ\text{C}$ and rate $r \in [-2, 2]\,^\circ\text{C/min}$. Let the controller output $u \in [0, 1]$ denote heater power as

a fraction of maximum. We use simple piecewise-linear labels:

$$\mu_{\text{Cold}}(e) = \begin{cases} 1, & e \le -5, \\ \frac{0-e}{5}, & -5 < e < 0, \\ 0, & e \ge 0, \end{cases} \qquad \mu_{\text{Comfy}}(e) = \max\left(0,\ 1 - \frac{|e|}{2.5}\right),$$

$$\mu_{\text{Stable}}(r) = \max(0,\ 1 - |r|)\,, \qquad \mu_{\text{Rising}}(r) = \begin{cases} 0, & r \le 0, \\ \frac{r}{2}, & 0 < r < 2, \\ 1, & r \ge 2. \end{cases}$$

For u, use the standard three-label output family (Low/Medium/High) from Chapter 16:

$$\mu_{\text{Low}}(u) = \max\left(0,\ 1 - \tfrac{u}{0.5}\right),$$
$$\mu_{\text{Medium}}(u) = \max\left(0,\ 1 - \tfrac{|u-0.5|}{0.25}\right),$$
$$\mu_{\text{High}}(u) = \max\left(0,\ \tfrac{u-0.5}{0.5}\right).$$

Rule base.

1. IF e is Cold AND r is Stable THEN u is High.
2. IF e is Cold AND r is Rising THEN u is Medium.
3. IF e is Comfy THEN u is Low.

Fuzzify inputs and compute firing strengths. For inputs $e = -2$ and $r = 0.5$,

$$\mu_{\text{Cold}}(-2) = 0.4, \quad \mu_{\text{Comfy}}(-2) = 0.2, \quad \mu_{\text{Stable}}(0.5) = 0.5, \quad \mu_{\text{Rising}}(0.5) = 0.25.$$

These come directly from the definitions: $\mu_{\text{Cold}}(-2) = (0 - (-2))/5 = 0.4$ and $\mu_{\text{Rising}}(0.5) = 0.5/2 = 0.25$. With $T = \min$ ((18.2)), the rule firing strengths are

$$\alpha_1 = \min(0.4, 0.5) = 0.4, \quad \alpha_2 = \min(0.4, 0.25) = 0.25, \quad \alpha_3 = 0.2.$$

Implication, aggregation, and defuzzification. Using Mamdani min-implication ((18.4)), each consequent is clipped at its firing strength (e.g.,

$\mu'_{\text{High}}(u) = \min(\alpha_1, \mu_{\text{High}}(u)))$. Aggregating consequents by max ((18.6)) yields $\mu_{\text{out}}(u)$. Finally, centroid defuzzification ((18.7)) produces a crisp heater-power command. Using a uniform grid on $u \in [0, 1]$ (step 10^{-4}) gives

$$u^\star \approx 0.577.$$

This value is above the midpoint because the *Cold* rules (High/Medium consequents) fire more strongly than the *Comfy* rule, while the positive rate term (*Rising*) tempers full-*High* output. In implementations, compute the centroid numerically (and log the grid/step size) rather than relying on heuristic convex combinations when overlap patterns become complex.

Pipeline at a glance (Mamdani/Larsen)

```
for each rule j:
    alpha_j = T( mu_A1(x1), ..., mu_An(xn) )
        # firing strength
    mu_Bprime_j(y) = implication(alpha_j, mu_Bj(y))
        # clip or scale
mu_out(y) = S( mu_Bprime_1(y), ..., mu_Bprime_R(y) )
    # aggregate (mu_out == mu_Bagg)
y_star = centroid(mu_out(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.

18.7 Mamdani vs. Sugeno/Takagi–Sugeno systems

Mamdani-style inference (clipped 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. Let r denote the rate-of-change ($r = \dot{e}$):

$$\text{IF } e \text{ is } A_i \text{ AND } r \text{ is } B_i \text{ THEN } u_i = p_i e + q_i r + r_i.$$

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 Stable 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]

[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.

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.

Minimum viable mastery.

- For a rule base, compute firing strengths, apply implication, aggregate consequents, and compute a centroid output.
- Explain the Mamdani vs. Sugeno/TSK tradeoff (interpretability vs. efficiency/tunability).
- Treat operator choice as part of the specification and keep it consistent across examples and implementations.

Common pitfalls.

- Defining memberships on incompatible scales (inputs never trigger, or everything triggers).
- Writing many redundant rules and then tuning symptoms rather than consolidating the rule base.
- Comparing controllers without stating operators, defuzzification, and evaluation protocol.

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.

If you are skipping ahead. When you reach Chapter 19, keep this chapter's audit instinct: log your design choices. Evolutionary and fuzzy systems both invite silent degrees of freedom that only become visible with disciplined reporting.

Where we head next. This trilogy treated fuzzy inference as a *behavioral* design tool: we copied the *form* of human reasoning explicitly (linguistic predicates, auditable operators, and rules you can inspect). Chapter 19 shifts to a different nature-inspired paradigm: instead of copying reasoning logic, we copy an *adaptation process* we associate with natural evolution (selection + variation under a fitness budget) and hope useful structure emerges from the search—think Karl Sims-style evolved behaviors as the intuition. The goal is not faithfulness to evolutionary theory; the goal is an algorithmic optimizer that can tune memberships, rule weights, and other discrete/continuous knobs when hand-tuning is brittle and evaluations are noisy, expensive, or constrained.

Part III takeaways

- Soft computing makes uncertainty explicit: degrees of membership and rule-based aggregation.
- Interpretability is a design variable: rules, operators, and defuzzification encode assumptions.
- Debugging is empirical: test edge cases, inspect rule firing, and validate monotonicity and scale.
- The same audit mindset applies: define failure modes up front and track them as you tune operators and rules.

Part IV: Evolutionary optimization

How to read Part IV (rolling window)

- **Optimization lens:** treat evolutionary algorithms as a budgeted optimizer for black-box, noisy, or constrained objectives (not as biology).
- **Repetition is intentional:** the GA loop appears as a big-picture story, then as operator details, then as a numeric trace you can verify in code.
- **Reporting is part of correctness:** match evaluation budgets, run multiple seeds, and report distributions and constraint handling explicitly.

19 Introduction to Evolutionary Computing

Parts I–III built three complementary toolkits: optimize models against data (ERM), learn representations with gradients, and encode auditable heuristics with fuzzy rules. Each is powerful when you can write a smooth objective or commit to a compact rule base. Many engineering choices, however, live outside that comfort zone: the knobs are discrete or tightly constrained, evaluations are

noisy or expensive, and the landscape is rugged enough that local improvement is unreliable.

Evolutionary computing treats that situation as search. Instead of following gradients or hand-tuning rules, we maintain a population of candidate designs, score them with a fitness function, and use selection plus variation to explore and refine solutions within a fixed budget. After the fuzzy trilogy (Chapters 15 to 18), this chapter develops the evolutionary strand introduced in Chapter 1 and focuses on population-based optimization under constraints, budgets, and stochastic noise.

A distinct paradigm (and what "nature-inspired" means here). Fuzzy inference systems are a behavioral modeling tool: we write down the reasoning logic directly so a human can audit why a particular decision was made. Evolutionary computing aims at a different target. It encodes *candidate designs* and uses a simplified selection–variation loop—an algorithmic echo of natural evolution—to search for designs that perform well under a fixed evaluation budget (the Karl Sims-style intuition: emergence via selection and variation, not hand-coded reasoning). The point is not to be faithful to evolutionary biology; the point is that this process can produce designs that respond well to situations under constraints and noise even when gradients are unavailable and the knobs are mixed discrete/continuous.

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.

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

Evolutionary computing is the strand you reach for when the objective is a black box (simulation, experiment, or expensive audit) or when the design space is a mix of continuous and discrete choices. The modeling contract stays simple: pick an encoding, define a fitness function, state the constraints, and commit to an evaluation budget. The algorithm then trades exploration (diversity) against exploitation (selecting what works) as it searches a rugged landscape.

In the fuzzy trilogy, once you commit to membership shapes and a rule base, a practical question remains: how do you tune those choices (and their trade-offs) when hand tweaks do not scale? Evolutionary search provides a disciplined answer: score candidate controllers, select the better ones, perturb/recombine them, and keep iterating until the improvements flatten or the budget runs out.

Vignette: budgeted search in the wild (locomotion, ML/LLMs, and physical experiments)

Robot gait / locomotion. Suppose we want a legged robot to move forward robustly across slightly different terrains. The design has mixed knobs: continuous parameters (gains, trajectory coefficients, penalty weights) and discrete choices (controller mode, safety logic, constraint-handling policy). A single trial is expensive and noisy (contact dynamics, initial conditions, sensor noise), so it is natural to treat performance as a black-box fitness, average over multiple seeds, and search under a strict evaluation budget.

Hyperparameter and pipeline tuning (including LLM systems). Many modern learning pipelines have the same structure: discrete design choices (model family, system configuration variant, data filters) plus continuous knobs (learning rates, thresholds, temperatures), measured by an empirical score with variance across runs. Evolutionary search is attractive when gradients are unavailable, the evaluation surface is rugged, or the objective invites silent failure modes (metric gaming, constraint violations).

Physical-world analogy (jet nozzle). Classic evolution-strategy examples describe optimizing a jet-nozzle shape via random mutations plus selection: you do not differentiate the wind tunnel; you spend a limited trial budget to evolve better designs. The point of the analogy is not biology, but budgeted optimization under constraints.

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, conver-

gence 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

Notation handoff

Vectors of candidate solutions are written as $\mathbf{x}$, objective values as $J(\mathbf{x})$, and population members as individuals/chromosomes depending on encoding. If symbol reuse from earlier chapters becomes ambiguous, default to local definitions and consult Appendix D.

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 $J : \mathbb{R}^n \to \mathbb{R}$. Formally, we want to solve

$$\mathbf{x}^* = \arg\min_{\mathbf{x} \in \mathcal{D}} J(\mathbf{x}), \tag{19.1}$$

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 J may be *non-convex*, exhibiting multiple local minima and maxima.
- The objective or constraints may be undefined or discontinuous in parts of the domain, breaking gradient assumptions.

- The feasible set may include combinatorial or integer constraints that resist continuous methods.
- There may be no closed-form or deterministic method to find the global optimum, especially for NP-hard variants.
- 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 J 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

Checkpoint: you now have the *why* for evolutionary search—rugged landscapes, constraints, discontinuities, and budgets that make local improvement unreli-

able. The rest of this chapter builds one canonical template in three passes: (i) a big-picture story (population + evaluation + variation), (ii) operator-level definitions (selection/crossover/mutation and constraint handling), and (iii) an implementation-facing sanity-check (flowchart, pseudocode, and a one-generation numeric trace). Read repeated mentions of "the GA loop" as *preview* → *formalization* → *checklist*, not as duplication.

Keep the "rugged landscape" picture in mind 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.

These issues compound the nonconvexity and constraint challenges above: discontinuities and combinatorial feasibility break gradient assumptions, and global guarantees are out of reach for many NP-hard variants. 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 Evolutionary Computing at a Glance

Evolutionary computing (EC) is best viewed as a *budgeted optimizer*: maintain a population of candidates, evaluate them with a fitness function, bias sampling toward better candidates (selection), generate variation (crossover/mutation), handle constraints, and replace the population. Repeat until a stopping rule triggers (budget, target quality, or no-improvement window).

Key Idea We evolve a *set* of candidate solutions over successive generations. Unlike gradient-based methods, evolutionary algorithms do not require differen-

tiability, and unlike brute force, they trade guarantees for tractable performance under noise, discontinuities, and combinatorial feasibility.

Genetic Algorithms (GAs) One of the most well-known evolutionary algorithms is the Genetic Algorithm (GA). GAs *loosely* mimic evolutionary ideas using a simplified, abstracted model of mechanisms such as selection, crossover, and mutation.

19.9 Biological Inspiration: Evolutionary Concepts

This section is terminology and motivation. If you already think of a GA as "selection + variation + replacement under a fitness budget," feel free to skim; the algorithmic details begin in Section 19.12.

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.
- **Replacement/elitism:** Form the next generation by replacing some individuals with offspring; optionally preserve the best solutions explicitly.
- **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

$$\mathbf{c} = (c_1, c_2, \ldots, c_L),$$

where each gene c_i encodes a particular trait or parameter, and L is the chromosome length.

Inheritance and Crossover During reproduction, offspring inherit genes from parents via crossover and mutation (with some genes passing through unchanged). The next figure gives an intuitive operator-level picture; full selection and crossover details follow in Section 19.17 and Section 19.18.

Figure 73 summarizes selection, crossover, and constraint handling at the operator level.

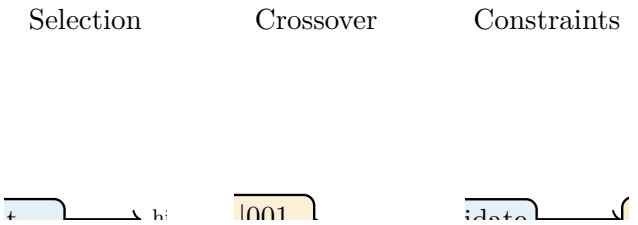

Figure 73: Evolutionary micro-operators. Left: fitter individuals get sampled more often (roulette/tournament). Middle: crossover splices parents by a mask (one-point shown). Right: constraint handling routes offspring through repair/penalty/feasibility before evaluation. Use this to map an implementation to the canonical operator loop.

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, and then applying mutation:

$$\mathbf{o} = \text{Mutate}\big(\text{Crossover}(\mathbf{p}_1, \mathbf{p}_2, \mathbf{u})\big). \tag{19.2}$$

The crossover operator selects which genes come from which parent, often modeled by a binary mask $\mathbf{u} \in \{0,1\}^L$, where

$$o_i = \begin{cases} (p_1)_i, & \text{if } u_i = 0, \\ (p_2)_i, & \text{if } u_i = 1. \end{cases}$$

Mutation (see Section 19.19) perturbs genes with a small probability p_m.

Fitness and selection (preview) Chromosomes encode candidate solutions (phenotypes), and fitness scores quantify how well each candidate meets the objective. For example, consider chromosomes representing facial feature variants with fitness values

$$f = \{80, 75, 60, 65, 40, 20\}.$$

Higher-fitness chromosomes should be sampled more often, but not deterministically: occasional selection of weaker candidates preserves diversity and reduces premature convergence. A simple baseline is roulette selection: normalize nonnegative fitness values into probabilities and sample with replacement. The point is the *bias* toward what works (not biological fidelity); we formalize selection and

work a roulette example in Section 19.17.

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 $g_j(\mathbf{x}) \le 0$ 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 recombination.

19.14 Encoding in Genetic Algorithms

We use the chromosome representation from Section 19.12; the focus here is how different encodings affect operators, constraints, and search efficiency.

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

$$0 < x \leq 15$$

For example, $x = 5$ gives $f(x) = 5/2 + 125/5 = 27.5$.

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 For minimization problems, selection still expects "higher is better," so define fitness as a monotone transform of the cost (e.g., $F(x) = -f(x)$ or $F(x) = 1/(1 + f(x))$), and then incorporate penalties or feasibility rules 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. In practice, these operators are the *control knobs* that decide how aggressively the search exploits what looks good versus how much it explores new regions of the design space.

Where operators sit in the GA loop. At a high level, each generation repeats the same pipeline:

1. Evaluate fitness (and apply penalties/feasibility rules if constrained).
2. Select parents (controls *selection pressure*).
3. Generate variation via crossover and mutation (controls *mixing* and *novelty*).
4. Form the next population (often with elitism or replacement rules).

The next subsections unpack these pieces: selection in Section 19.17, crossover in Section 19.18, and mutation in Section 19.19.

19.17 Selection in Genetic Algorithms

Selection answers the question: *who gets to reproduce?* It is the main mechanism that biases sampling toward better candidates, and it therefore determines the algorithm's *selection pressure*. Too much pressure collapses diversity and risks premature convergence; too little reduces the GA to near-random search. In this section we formalize how fitness values translate into parent choice (roulette-style probabilities), then compare alternatives (ranking and tournament selection) that are often more stable when raw fitness values are poorly scaled. The goal is to connect the big-picture GA loop to implementable parent selection rules you can audit and tune.

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.

Practical note (fitness scaling). Proportional (roulette) selection assumes *nonnegative* fitness values. If your objective can be negative (as in Table 10) or you are minimizing a cost, define a shifted/scaled fitness, e.g., $\tilde{f}_i = f_i - \min_j f_j + \varepsilon$ with $\varepsilon > 0$, or use rank/tournament selection, which only relies on order comparisons.

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:

$$p = [0.10, 0.20, 0.05, 0.15, 0.50].$$

For instance, $p_2 = 20/100 = 0.20$ and $p_5 = 50/100 = 0.50$. 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 formalizes the intuition previewed above and supports controlled exploration.

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 material between parents to create novel combinations.

Multi-point and uniform crossover. Two-point or multi-point crossover swaps multiple segments between parents; uniform crossover swaps each gene independently with a fixed probability. These variants increase mixing but can disrupt building blocks if overused.

Crossover probability. Apply crossover with probability p_c (typically 0.6–0.9). When crossover is skipped, offspring are usually copied forward and then mutated, which helps preserve good structures while maintaining exploration.

19.19 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 binary chromosome $\mathbf{c} \in \{0,1\}^L$, mutation produces $\mathbf{c}'$ by mutating each bit c_i independently with probability p_m:

$$c_i' = \begin{cases} 1 - c_i, & \text{with probability } p_m, \\ c_i, & \text{with probability } 1 - p_m. \end{cases}$$

19.20 Summary of Genetic Operators and Their Probabilities

You have now seen each operator in isolation. When you implement a GA, you typically configure the loop with a small set of *operator knobs*:

- **Selection scheme + pressure:** roulette/ranking/tournament selection, plus an explicit pressure parameter (e.g., fitness scaling, rank parameter s, tournament size) and often an elitism rate (top k or top $e\%$ copied forward).
- **Crossover probability p_c:** with probability p_c, recombine two parents; otherwise copy parents forward (and still allow mutation).
- **Mutation probability p_m:** for binary encodings, mutate each bit independently with probability p_m (a common default is $p_m \approx 1/L$ for length L); for real encodings, p_m usually corresponds to a mutation *scale* (e.g., Gaussian noise standard deviation) rather than bit flips.

To keep notation compact later, we write these scheme-dependent settings abstractly as

$$\begin{aligned} p_s &\equiv \text{selection pressure parameter(s) (scheme-dependent)}, \\ p_c &= \text{probability of applying crossover}, \\ p_m &= \text{mutation probability (per gene/bit, unless stated otherwise)}. \end{aligned}$$

Tuning these settings controls exploration versus exploitation and strongly affects whether the algorithm stagnates early or continues to make progress.

19.21 Known Issues in Genetic Algorithms

Risk & audit

- **Premature convergence:** aggressive selection or weak mutation collapses diversity before good regions are explored.
- **Budget mismatch:** comparisons are invalid unless algorithms share evaluation budgets and stopping rules.
- **Constraint leakage:** hidden infeasibility can inflate scores; use explicit feasibility checks in logs and plots.
- **Single-run overclaim:** stochastic algorithms require multi-seed reporting with spread, not one best trajectory.
- **Objective gaming:** fitness proxies can drift from deployment goals; audit secondary metrics and failure cases.

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.

19.22 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

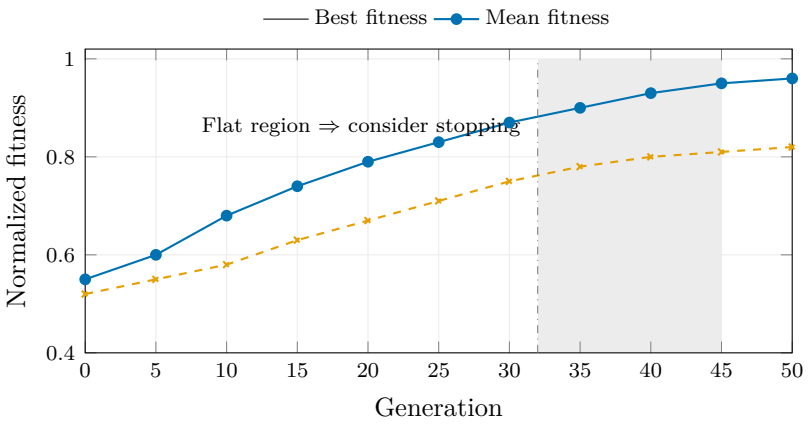

Figure 74: Illustrative GA run showing the best and mean normalized fitness over 50 generations. Flat regions motivate "no improvement" stopping rules, while steady separation between best and mean indicates ongoing selection pressure. Helpful for diagnosing premature convergence versus ongoing exploration.

condition is met first. Figure 74 visualizes such a run, making it easy to spot plateaus and the persistent gap between best and mean fitness.

19.23 Summary of Genetic Algorithm Workflow

This is the implementation-facing assembly of the GA loop. Earlier sections motivated the need for population-based search and defined each operator; here we put them back together and give three complementary sanity-check views: Figure 75 for control flow, Table 10 for a one-generation numeric trace, and Section 19.24 for a minimal implementation template. All three describe the same loop at different levels of detail.

For the toy generation in Table 10, fitness values are computed using $f(x) = \cos(5\pi x)\exp\left(-x^2\right)$ from Section 19.25.

Table 10 is the step-by-step toy generation trace used to ground the operator sequence.

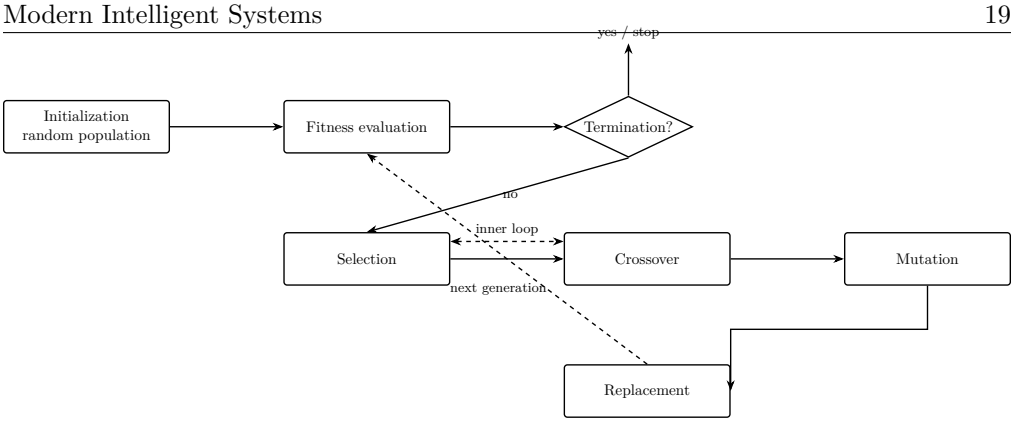

Figure 75: GA flowchart showing the iterative process: initialization leads to fitness evaluation and a termination check. If not terminated, the algorithm proceeds through selection, crossover, mutation, and replacement, which then feeds the next generation's fitness evaluation. Reference when verifying that your implementation preserves the intended control flow.

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	0.553, -0.377, 0.177, 0.687
Select parents (example)	0010_2, 0101_2	0.125, 0.3125	-0.377, 0.177
Crossover (one-point)	Parents: 00|10, 01|01 $\rightarrow$ offspring: 00|01, 01|10	0.0625, 0.3750	0.553, 0.803
Mutation (flip one bit)	$0110_2 \rightarrow 0111_2$	0.4375	0.687

Table 10: Toy GA generation on a bounded interval. One crossover and mutation illustrate how the fitness function guides selection before the next generation. Use this to explain how variation operators interact with selection pressure.

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.

19.24 Pseudocode Representation

The GA can be expressed in pseudocode as follows. Treat this as the "minimum viable" skeleton; a production implementation additionally records random seeds, evaluation budgets, constraint handling decisions, and per-generation

diagnostics (best/mean fitness, diversity) so results are reproducible and interpretable.

```
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
```

19.25 Example: GA for a Constrained Optimization Problem

Consider the problem of *maximizing* 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: Fixed-point with resolution 0.001: store an integer $n \in \{0, 1, \ldots, 500\}$ and decode via $x = n/1000$ (a 9-bit representation covers 0–511)
- Crossover probability: 25%
- Mutation probability: 10%
- Selection: Truncation selection (example): select the top five chromosomes by fitness as parents each generation

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 and treat it as a fitness score (higher is better). If instead you are minimizing a cost, convert it to fitness via a monotone transform (e.g., $-J(x)$, a shifted score, or a rank-based scheme) so that selection still prefers better candidates.

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, if the initial population contains

$$x \in \{0.04,\ 0.09,\ 0.13,\ 0.18,\ 0.22,\ 0.27,\\ 0.31,\ 0.36,\ 0.42,\ 0.48\},$$

then the corresponding objective values are

$$f(x) \in \{0.808,\ 0.155,\ -0.446,\ -0.921,\ -0.906,\ -0.422,\\ 0.142,\ 0.711,\ 0.797,\ 0.245\},$$

rounded to three decimals. Each chromosome uses a 9-bit fixed-point code (3 fractional digits), decoded by interpreting the bits as an integer n and scaling via $x = n/1000$. Any decoded values outside [0, 0.5] are repaired (e.g., clipped to bounds); note that $x = 0$ is allowed in this example.

A single generation could proceed as follows:

- Select the top five chromosomes by fitness.
- Apply one-point crossover on the 9-bit fixed-point codes (MSB-first).

For example, $0.203 \mapsto 011001011_2$ and $0.359 \mapsto 101100111_2$; cutting after 5 bits and swapping tails yields offspring $011000111_2 \mapsto 0.199$ and $101101011_2 \mapsto 0.363$.

- 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, following Appendix E as the default reporting template. 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.

19.26 Genetic Algorithms: Iterative Process and Convergence

This section focuses on convergence behavior; the GA cycle itself is summarized in Section 19.23 and the control flow in Figure 75. Over generations, populations tend to cluster around better regions of the fitness landscape; flat best-fitness curves signal stagnation, while a persistent gap between best and mean indicates ongoing selection pressure (see Figure 74). Convergence typically occurs after a problem-dependent number of generations, and the best-so-far solution should be treated as a high-quality approximation rather than a guaranteed global optimum.

19.27 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.

Evolution strategies (ES): step-size adaptation and self-adaptation

Evolution strategies are a real-coded evolutionary family that makes one engineering idea explicit: *mutation step size is part of the algorithm's state, not a fixed constant.* In a simple isotropic form,

$$\mathbf{x}' = \mathbf{x} + \boldsymbol{\epsilon}, \qquad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0},\, \sigma^2 \mathbf{I}),$$

where σ controls exploration (large σ) versus local refinement (small σ). A classic adaptive heuristic is the "1/5 success rule": if more than 20% of recent mutations improve fitness, increase σ; if fewer than 20% succeed, decrease σ. Self-adaptive variants go one step further by embedding σ (and sometimes covariance structure) into the chromosome itself so selection pressure co-evolves the search behavior alongside the solution parameters.

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.

19.28 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 algorithms by evolving program structures (often trees) rather than fixed-length chromosomes.

19.29 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 metaheuristics; the earlier sections already define selection, crossover, mutation, and the loop in detail. For recap, keep the design levers in view and revisit the worked traces when needed:

- **Representation:** How solutions are encoded (bit strings, real vectors, trees) and what constraints must be preserved.
- **Fitness and evaluation:** What you reward, how you handle noisy measurements, and how constraints enter the score.
- **Operator/loop settings:** Selection pressure, crossover/mutation rates, and stopping criteria.

See Section 19.23 for control flow, Table 10 for a numeric generation trace, and Sections 19.17 to 19.19 for operator details.

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.

19.30 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 objec-

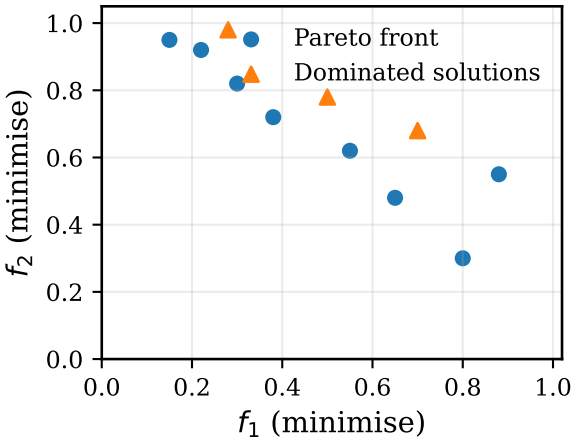

Figure 76: Sample Pareto front for two objectives. NSGA-II keeps all non-dominated points (blue) while pushing dominated solutions (orange) toward the front via selection, yielding a spread of trade-offs in one run. Use this when interpreting multi-objective results as trade-offs, not a single optimum.

tives into weighted subproblems; SPEA2/IBEA are popular alternatives. Always plot the Pareto set and budget-matched hypervolume traces when comparing algorithms.

Figure 76 is the trade-off view used to interpret multi-objective runs.

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.

Minimum viable mastery.

- Specify genotype, variation operators, selection scheme, and termination criteria in a way that is reproducible.
- Distinguish constraint handling strategies and justify the one used (penalty vs. repair vs. feasibility rules).
- Report multi-objective results as trade-offs (Pareto fronts) rather than as a single scalar score.

Common pitfalls.

- Over-interpreting a single stochastic run (report multiple seeds and dispersion).
- Using aggressive selection and low mutation, collapsing diversity and getting stuck in deceptive basins.
- Comparing algorithms without matching evaluation budget (fitness calls), constraint handling, and stopping rules.

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.

If you are skipping ahead. This chapter is largely self-contained. If you revisit earlier model chapters, keep the common thread in mind: every method still requires a clear objective, a diagnostic that detects failure, and a reporting protocol that survives replication.

Where we head next. This chapter closes the soft-computing thread. For targeted refreshers, Appendix A summarizes linear-systems prerequisites and Appendix B consolidates kernel-method context used earlier in the book.

Part IV takeaways

- Evolutionary search is an optimization tool when gradients are unavailable or unreliable.
- Operators (selection, crossover, mutation) encode exploration vs. exploitation; tune them against variance across runs.
- Constraints and multi-objectives are first-class: define feasibility and trade-offs before optimizing.
- Report reproducibly: seeds, multiple runs, and distributions matter more than a single best trajectory.

Back matter

Key Takeaways

Chapter 1 *About This Book* sets a reading strategy, summarizes notation/figure conventions, and motivates the four-level taxonomy for systems thinking across chapters.

Chapter 2 *Symbolic Integration and Problem-Solving Strategies* frames safe substitutions, heuristic branches, and numeric fallbacks inside a transformation tree for 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* builds probabilistic classifiers using the Chapter 3 toolkit, emphasizes ROC/PR analysis, and treats 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 catalogs 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 earlier diagnostics.

Chapter 13 *NLP Pipelines and Responsible Deployment* links static/contextual embeddings to downstream tasks, adds bias/calibration checklists, and closes with a deployment-readiness assessment.

Chapter 14 *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 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 Mamdani and Sugeno pipelines (aggregation, implication, defuzzification) and studies 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 14), 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 14), and self-adaptive evolutionary strategies (Chapter 19). These examples illustrate early steps toward systems that refine their own policies.

Figure 77 provides the visual map; Table 11 serves as a compact lookup for model-family placement across levels and learning signals.

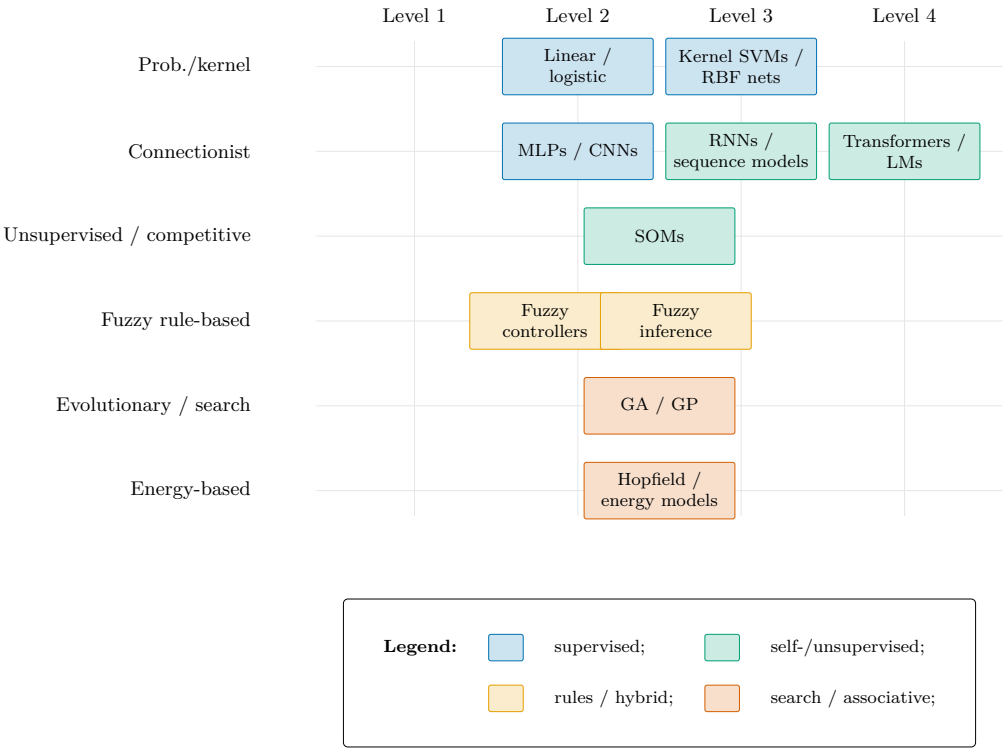

Figure 77: Color-coded map of model families across agent level, model nature, and learning signal, highlighting where each family sits in the taxonomy and where to read next.

Table 11: 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. The table serves as a compact lookup for model-family placement across levels and learning signals.

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. doi: 10.1103/physrevlett.55.1530.

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety. *arXiv preprint*, 2016.

Ronald C. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR)*, 2015.

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. doi: 10.1016/s0020-7373(80)80055-1.

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. doi: 10.1073/pnas.1903070116.

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 (NeurIPS)*, 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 (NeurIPS)*, 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. doi: 10.1109/jra.1986.1087032.

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 *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 103–111, 2014. doi: 10.3115/v1/w14-4012.

Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995. doi: 10.1023/a:1022627411411.

M. Cottrell and J. C. Fort. A stochastic model of retinotopy: A self organizing process. *Biological Cybernetics*, 53(6):405–411, 1986. doi: 10.1007/bf00318206.

Michael T. Cox and A. Raja. Metareasoning: A manifesto. *AI Magazine*, 32(1):39–54, 2011. doi: 10.1609/aimag.v32i1.2304.

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 (NeurIPS)*, 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. URL `https://www.complex-systems.com/abstracts/v09_i02_a02/`.

Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002. doi: 10.1109/4235.996017.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 4171–4186, 2019.

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. doi: 10.1207/s15516709cog1402_1.

E. Erwin, K. Obermayer, and K. Schulten. Self-organizing maps: Ordering, convergence properties and energy functions. *Biological Cybernetics*, 67(1): 47–55, 1992. doi: 10.1007/bf00201801.

Bernd Fritzke. A growing neural gas network learns topologies. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 7, pages 625–632, 1994. URL `https://proceedings.neurips.cc/paper/1994/hash/1ccefb98e3ce8f4d1536b8f2f6f7f15a-Abstract.html`.

Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 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*, 2023.

Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. On calibration of modern neural networks. In *International Conference on Machine Learning (ICML)*, 2017.

Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001. doi: 10.1162/106365601750190398.

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. doi: 10.1162/neco.1997.9.8.1735.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, et al. Training compute-optimal large language models. In *arXiv preprint*, 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. doi: 10.1073/pnas.79.8.2554.

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*, 2020.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations (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. doi: 10.1007/bf00337288.

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 (NeurIPS)*, 25:1097–1105, 2012.

Dmitry Krotov and John J. Hopfield. Dense associative memory for pattern recognition. *Advances in Neural Information Processing Systems (NeurIPS)*, 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 (ICLR)*, 2017.

Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2014.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *International Conference on Learning Representations (ICLR)*, 2019.

J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967. URL `https://projecteuclid.org/euclid.bsmsp/1200512992`.

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. doi: 10.1016/s0020-7373(75)80002-2.

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. doi: 10.1109/72.238311.

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. doi: 10.1007/bf02478259.

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. doi: 10.1109/tit.1987.1057328.

Charles A. Micchelli. Interpolation of scattered data: Distance matrices and conditionally positive definite functions. *Constructive Approximation*, 2(1): 11–22, 1986. doi: 10.1007/bf01893414.

Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Černocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Interspeech*, 2010. doi: 10.21437/interspeech.2010-343.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *International Conference on Learning Representations (ICLR)*, 2013.

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. doi: 10.1162/neco.1991.3.2.246.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014. doi: 10.3115/v1/d14-1162.

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. doi: 10.1109/5.58326.

David Poole and Alan Mackworth. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, 2nd ed., 2017.

Ofir Press, Noah A. Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *Advances in Neural Information Processing Systems (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 (ICLR)*, 2021.

Robert H. Risch. The problem of integration in finite terms. *Transactions of the American Mathematical Society*, 139:167–189, 1969. doi: 10.2307/1995313.

Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. doi: 10.1037/h0042519.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986. doi: 10.1038/323533a0.

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 *International Conference on Computational Linguistics (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 (ICLR)*, 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. doi: 10.1023/a:1008202821328.

Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2021.

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. doi: 10.1109/tsmc.1985.6313399.

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 (NeurIPS)*, volume 30, 2017.

Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. In *1960 IRE WESCON Convention Record*, volume 4, pages 96–104, 1960. doi: 10.21236/ad0241531.

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. doi: 10.1098/rspb.1976.0087.

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. doi: 10.1016/S0019-9958(65)90241-X.

Lotfi A. Zadeh. The concept of a linguistic variable and its application to approximate reasoning. *Information Sciences*, 8(3):199–249, 1975. doi: 10.1016/0020-0255(75)90036-5.

Lotfi A. Zadeh. Fuzzy logic, neural networks, and soft computing. *Communications of the ACM*, 37(3):77–84, 1994. doi: 10.1145/175247.175255.

Lotfi A. Zadeh. What is soft computing? *Soft Computing*, 1:1–2, 1997. No canonical DOI found in Crossref; keep journal metadata authoritative.

Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm. *Technical Re-*

port 103, 2002. URL https://tik-old.ee.ethz.ch/db/public/tik/?db=publications&form=report_single_publication&publication_id=1287.

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.

If you are not taking ECE 657: you can skip this appendix entirely.

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.

Reading note. When a symbol is reused with a different meaning, chapters typically flag the local meaning near the first use. When reading out of order, use this index to disambiguate quickly and keep the local convention consistent.

E Reproducibility and Reporting Standards

This appendix defines the minimum reporting standard used throughout this book when experiments include stochastic training, model selection, or constrained optimization. The objective is simple: results should be auditable and repeatable by another reader with the same code and data access.

Core reporting template (required fields)

- **Data protocol:** split policy, leakage controls, and preprocessing/tokenization version.
- **Model protocol:** architecture configuration, parameter count, initialization policy, and regularization settings.
- **Optimization protocol:** optimizer, schedule, batch size, stopping rule, and checkpoint selection criterion.
- **Evaluation protocol:** primary metric, calibration/slice checks, and whether thresholds were tuned on validation only.
- **Variance protocol:** random seeds, number of runs, and summary statistics (median/mean plus spread).

Minimum acceptable evidence

- Report at least 5 seeds for low-cost experiments and 10+ seeds when claims depend on small metric differences.
- Show both central tendency and spread (e.g., median + interquartile range, or mean + standard deviation).
- Match evaluation budgets when comparing methods (same number of epochs/fitness calls and same stopping policy).
- Preserve a machine-readable experiment log (configuration + metrics + seed) for each run.

Common failure modes to avoid

- **Best-run reporting:** publishing only the strongest seed and omitting dispersion.
- **Budget mismatch:** giving one method more training/evaluation budget than another.
- **Selection leakage:** tuning hyperparameters on test data (explicitly or implicitly).

- **Unversioned preprocessing:** changing tokenization or normalization without recording the revision.

Chapter-specific notes

- **Supervised/Logistic chapters:** pair accuracy-like metrics with calibration and slice checks.
- **Deep-learning chapters:** track gradient health, early stopping criteria, and checkpoint restore policy.
- **Evolutionary chapter:** report evaluation-budget-normalized performance and multi-seed front dispersion.

Practical rule. If a claim changes a design decision, it must include enough protocol detail for a third party to reproduce the comparison.