

ML Lecture Notes: Week 1

Intellectual History of AI and Machine Learning

Week Overview

This week surveys the intellectual history of AI and machine learning, from philosophical origins to the current deep learning era. The goal is not merely historical—understanding *why* certain approaches succeeded or failed provides insight into current methods and their limitations.

Key questions: Why did neural networks “work” in 2012 when they didn’t in 1969? What is the relationship between symbolic and connectionist approaches? Why do we call it “learning”?

1 Philosophical Foundations: The Computational Theory of Mind

Section Summary

This section traces how the ancient idea that “thought is computation” was formalised mathematically. Key figures: Hobbes (reasoning as symbol manipulation), Leibniz (universal calculus of thought), Turing (formalisation of computation itself). The Church-Turing thesis establishes what can be computed; Turing’s test operationalises machine intelligence.

The idea that reasoning can be mechanised has deep philosophical roots. Thomas Hobbes (1588–1679) argued that “reason is nothing but reckoning”—that thinking is fundamentally a form of computation over symbols. This radical claim suggests that the seemingly mysterious process of human thought might be reducible to mechanical operations, not unlike arithmetic. If true, it implies that brains *compute*, and what computes can, in principle, be approximated or replicated by other computing devices.

Gottfried Wilhelm Leibniz (1646–1716) extended this vision, dreaming of a *calculus ratiocinator*: a universal logical calculus that could resolve all disputes through calculation. Leibniz imagined that when two philosophers disagreed, they could simply say “Let us calculate!” and arrive at the correct answer through symbolic manipulation. While this vision proved overly optimistic (Gödel’s incompleteness theorems would later show its fundamental limitations), it planted the seed for formal logic and, eventually, computer science.

These ideas laid the groundwork for what philosophers now call the **computational theory of mind**—the hypothesis that cognition is fundamentally information processing, and that mental states can be understood as computational states operating over internal representations. On this view, the brain is a kind of biological computer, and thoughts are programs running on neural hardware.

Key Insight

If the mind is computational, then in principle it can be replicated in a machine. This philosophical stance underpins the entire AI enterprise. Note that this is a substantive empirical claim, not a logical necessity—and it remains contested among philosophers and cognitive scientists.

The mathematical formalisation came in the 20th century. Alan Turing's 1936 paper "On Computable Numbers" introduced the *Turing machine*—a theoretical model of computation that showed anything "effectively calculable" could be computed by a simple mechanical process. This established three profound results:

1. A precise definition of what it means to compute
2. The universality of computation (a single machine can simulate any other)
3. The limits of computation (some problems are *undecidable*—no algorithm can solve them)

Turing's 1950 paper "Computing Machinery and Intelligence" posed the question: *Can machines think?* Rather than debating definitions (what does "think" even mean?), Turing proposed an operational test—the **Turing Test**—where a machine passes if a human interrogator cannot reliably distinguish it from a human through text-based conversation. This behaviourist approach sidesteps metaphysical debates about consciousness and focuses on observable capabilities.

The Church-Turing Thesis

Any function that can be computed by an "effective procedure" (an algorithm) can be computed by a Turing machine. This is not a theorem but a *thesis*—it cannot be proven because "effective procedure" is an informal notion that predates its formalisation. However, every proposed formalisation of computation (lambda calculus, recursive functions, register machines, cellular automata) has been shown equivalent to Turing machines, lending strong empirical support to the thesis.

The thesis has profound implications: if it is correct, then Turing machines capture the full extent of what can be computed—there is no "super-computation" beyond their reach. Any computer, from a smartphone to a supercomputer, can compute exactly the same class of functions (though with vastly different speed and memory constraints).

Turing Machines: The Formal Model

A Turing machine consists of:

- An infinite tape divided into cells, each containing a symbol from a finite alphabet Γ
- A head that reads/writes symbols and moves left or right
- A finite set of states Q , including distinguished start and halt states
- A transition function $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$

Unpacking the transition function: Given the current state $q \in Q$ and the symbol $s \in \Gamma$ under the head, the function $\delta(q, s) = (q', s', d)$ specifies:

- s' : the new symbol to write in the current cell
- $d \in \{L, R\}$: the direction to move the head
- q' : the new state to enter

The machine operates deterministically: given current state and symbol, there is exactly one action to take. Computation proceeds until reaching a halt state (or running forever if no halt state is reached).

A **Universal Turing Machine** (UTM) can simulate any other Turing machine given its description as input—this is the theoretical foundation of the stored-program computer. The UTM reads a description of machine M and input x , then simulates what M would do on x . Your laptop is essentially a physical approximation of a UTM.

Connection to ML: Neural networks are often described as “universal function approximators.” This is a *different* sense of universality—they can approximate any continuous function to arbitrary precision (given sufficient width/depth), but they are still computed by (finite implementations of) Turing machines. The distinction between *computing* a function exactly and *approximating* it is crucial: Turing machines compute discrete functions exactly; neural networks approximate continuous functions to within ϵ .

2 Cybernetics and Early Neural Models (1940s–1950s)

Section Summary

The first computational models of neurons emerged from interdisciplinary work in neuroscience, mathematics, and engineering. McCulloch-Pitts neurons showed neural computation could implement Boolean logic. Hebb’s learning rule proposed how connections strengthen through use. Cybernetics provided the framework of feedback and self-regulation that would later inform control-theoretic views of learning.

Cybernetics, founded by Norbert Wiener in his 1948 book of the same name, studied systems that regulate themselves through feedback loops. The name comes from the Greek *kybernetes* (steersman), reflecting the core metaphor: a helmsman constantly adjusts the rudder based on the ship’s deviation from course. The key insight was that goal-directed behaviour emerges from systems that sense their environment and adjust their actions to reduce the error between current

and desired states.

Cybernetic Principles

- **Feedback:** Systems sense their outputs and adjust inputs accordingly (negative feedback reduces error; positive feedback amplifies it)
- **Homeostasis:** Tendency toward stable equilibrium states through self-regulation
- **Information:** “Differences that make a difference” (Gregory Bateson’s formulation)—information is defined by its effects on the system’s behaviour

These principles unified thinking about biological organisms, machines, and social systems under a common mathematical framework.

Cybernetics was genuinely interdisciplinary, bringing together engineers, mathematicians, neurophysiologists, and social scientists. The Macy Conferences (1946–1953) were legendary gatherings where figures like Wiener, John von Neumann, Warren McCulloch, and Margaret Mead discussed feedback, communication, and control across domains.

In 1943, Warren McCulloch (a neurophysiologist) and Walter Pitts (a mathematical prodigy) published “A Logical Calculus of Ideas Immanent in Nervous Activity”, proposing that neurons could be modelled as logical gates. Their **McCulloch-Pitts neuron** was a binary threshold unit: it fires (outputs 1) if the weighted sum of its inputs exceeds a threshold, otherwise it remains silent (outputs 0).

McCulloch-Pitts Neuron

A neuron j with n binary inputs x_1, \dots, x_n , weights w_1, \dots, w_n , and threshold θ computes:

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

Breaking this down:

- Each input $x_i \in \{0, 1\}$ represents whether a presynaptic neuron is firing
- Each weight w_i represents the strength of the synaptic connection (positive for excitatory, negative for inhibitory)
- The sum $\sum_{i=1}^n w_i x_i$ is the total “evidence” for firing
- The threshold θ is the decision boundary: fire if evidence exceeds threshold

This is a step function applied to a linear combination of inputs. McCulloch and Pitts showed that networks of such neurons could compute any Boolean function—AND, OR, NOT, and therefore any logical proposition. This established a deep connection between neural activity and symbolic logic, suggesting that the brain might literally implement logical reasoning.

Example implementations:

- **AND gate:** Set $w_1 = w_2 = 1$ and $\theta = 2$. Both inputs must be 1 for the sum to reach threshold.
- **OR gate:** Set $w_1 = w_2 = 1$ and $\theta = 1$. Either input being 1 suffices.
- **NOT gate:** Set $w_1 = -1$ and $\theta = 0$. The output is 1 only when the input is 0.

By composing these primitives, any Boolean function is realisable.

Limitations: The model was highly idealised. Real neurons have continuous-valued outputs, complex temporal dynamics (refractory periods, spike timing), analogue behaviour, and learning capabilities not captured by fixed weights. But as a *proof of concept* that neural-like systems could perform computation, it was profoundly influential.

Donald Hebb's 1949 book *The Organization of Behavior* proposed a learning rule based on a simple principle: “Neurons that fire together, wire together.” More precisely, if neuron A repeatedly participates in firing neuron B, the synaptic connection from A to B is strengthened. This **Hebbian learning** suggested how associations could be learned through experience—it provided a biological mechanism for memory and learning that did not require an external teacher.

Hebbian Learning: Formalisation

In modern notation, Hebb's principle becomes the update rule for the weight w_{ij} from neuron i to neuron j :

$$\Delta w_{ij} = \eta \cdot x_i \cdot y_j$$

where:

- $\eta > 0$ is a learning rate controlling how much weights change per update
- x_i is the presynaptic activity (the “sending” neuron’s output)
- y_j is the postsynaptic activity (the “receiving” neuron’s output)

The weight increases when both neurons are active simultaneously. This is an *unsupervised, local* learning rule: each synapse updates based only on the activity of its pre- and post-synaptic neurons, with no global error signal required.

Problem: Pure Hebbian learning has no mechanism for weights to decrease, leading to unbounded growth. If weights can only increase, the network eventually saturates. Modern variants address this:

- **Oja’s rule:** Adds weight decay proportional to the squared output, performing online PCA
- **BCM theory:** Introduces a sliding threshold that depends on recent activity

Connection to modern ML: Hebbian learning is a form of correlation-based learning. It appears in:

- Principal Component Analysis (PCA) via Oja’s rule
- Hopfield networks (symmetric Hebbian weights store associative memories)
- Contrastive Hebbian learning in Boltzmann machines
- Spike-timing-dependent plasticity (STDP) in computational neuroscience

The modern emphasis on gradient-based learning (backpropagation) is fundamentally different: it requires a global error signal propagated backwards through the network.

3 The Birth of Artificial Intelligence (1956)

Section Summary

The Dartmouth Workshop (1956) formally established AI as a field. The founding vision was ambitious: simulate any aspect of intelligence. Early symbolic AI achieved impressive results on constrained problems (theorem proving, simple games) but struggled with real-world complexity. The tension between symbolic and connectionist approaches—rules versus learning—was present from the start.

The term “Artificial Intelligence” was coined at the **Dartmouth Workshop** in the summer of 1956, organised by John McCarthy (who invented the term), Marvin Minsky, Nathaniel Rochester

(IBM), and Claude Shannon (father of information theory). The funding proposal stated:

“The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it.”

This bold conjecture—that *all* aspects of intelligence are simulable—marked the formal birth of AI as a field. The workshop brought together researchers who would dominate the field for decades, establishing the optimistic, ambitious tone that would characterise early AI.

Note the attendee **Claude Shannon**, the founder of information theory. His 1948 paper “A Mathematical Theory of Communication” introduced entropy as a measure of uncertainty—a concept now central to ML. Shannon’s entropy $H(X) = -\sum_x p(x) \log p(x)$ appears throughout machine learning: in cross-entropy loss functions for classification, in information-theoretic bounds on learning, and in maximum entropy models. His participation reflects the deep connections between information theory and AI from the field’s inception.

The early approach was predominantly **symbolic AI** (also called GOFAI—Good Old-Fashioned AI): intelligence as manipulation of symbolic representations according to formal rules. Programs would represent knowledge as logical statements and derive conclusions through inference. The **Physical Symbol System Hypothesis** (Newell & Simon) claimed that “a physical symbol system has the necessary and sufficient means for general intelligent action.”

Key Timeline: Birth of AI

- **1943**: McCulloch-Pitts neural model
- **1948**: Wiener’s *Cybernetics* published; Shannon’s information theory
- **1949**: Hebb’s learning rule
- **1950**: Turing’s “Computing Machinery and Intelligence”
- **1956**: Dartmouth Workshop—AI named as a field
- **1957**: Rosenblatt’s Perceptron
- **1958**: McCarthy creates LISP, the language of AI

Early successes demonstrated that machines could perform tasks previously thought to require intelligence:

- **Logic Theorist** (Newell & Simon, 1956): Proved 38 of the first 52 theorems from *Principia Mathematica*, finding a more elegant proof for one theorem than Russell and Whitehead had
- **General Problem Solver** (Newell & Simon, 1959): Attempted domain-general reasoning through means-ends analysis
- **ELIZA** (Weizenbaum, 1966): Early chatbot using pattern matching to simulate a Rogerian therapist—people found it surprisingly engaging, which disturbed its creator

The early AI community was characterised by bold predictions and optimism. Herbert Simon predicted in 1957 that within ten years, a computer would be world chess champion and prove an important new mathematical theorem. Marvin Minsky predicted in 1967 that “within a generation... the problem of creating ‘artificial intelligence’ will substantially be solved.” These predictions proved premature by decades—chess took until 1997, and “solving AI” remains elusive.

NB!

[The Pattern of Overpromising] The pattern of overconfident predictions is a recurring theme in AI history. Early researchers underestimated:

- The difficulty of common-sense reasoning (**Moravec's paradox**: hard problems are easy, easy problems are hard—chess is easier than walking)
- The importance of embodied, situated cognition
- The computational resources required
- The brittleness of systems outside narrow domains

Understanding this history should temper both excessive hype and excessive pessimism about current AI capabilities.

4 The Perceptron and Supervised Learning

Section Summary

The perceptron (Rosenblatt, 1957) introduced *learning from data*—adjusting weights based on errors. It formalised supervised learning and proved convergence guarantees for linearly separable problems. The perceptron's limitations (inability to learn XOR) would later prove pivotal, but its core ideas—weighted sums, thresholds, error-driven updates—remain foundational.

Frank Rosenblatt introduced the **Perceptron** in 1957—the first neural network that could *learn* from data. Unlike the fixed McCulloch-Pitts networks, perceptrons adjusted their weights based on errors, implementing what we now call supervised learning.

Supervised Learning: Formal Definition

Given a training set $\mathcal{D} = \{(x_1, y_1), \dots, (x_n, y_n)\}$ where $x_i \in \mathcal{X}$ are inputs and $y_i \in \mathcal{Y}$ are labels (provided by a “supervisor”), the goal is to learn a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ that generalises well to unseen data from the same distribution.

Unpacking the components:

- \mathcal{X} is the *input space*—the set of all possible inputs (e.g., \mathbb{R}^d for d -dimensional feature vectors)
- \mathcal{Y} is the *output space*—the set of all possible labels
- Each pair (x_i, y_i) is a *training example*: an input with its correct label
- “Generalises well” means performs accurately on new data not seen during training

Classification: \mathcal{Y} is discrete (e.g., $\{0, 1\}$ for binary classification, or $\{1, \dots, K\}$ for K -class classification)

Regression: $\mathcal{Y} = \mathbb{R}$ (or \mathbb{R}^d for multivariate regression)

The learning process typically minimises an empirical risk over the training data:

$$\hat{f} = \arg \min_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^n L(f(x_i), y_i)$$

Breaking this down:

- $L(\hat{y}, y)$ is a *loss function* measuring how bad prediction \hat{y} is when the true label is y
- $\frac{1}{n} \sum_{i=1}^n L(f(x_i), y_i)$ is the *empirical risk*—average loss over training data
- \mathcal{F} is the *hypothesis class*—the set of functions the learner considers
- The choice of \mathcal{F} embodies our *inductive bias*—our assumptions about what kinds of functions are likely to be correct

The Perceptron Algorithm

For a binary classification problem with $x \in \mathbb{R}^d$ and $y \in \{-1, +1\}$:

Model: The perceptron computes a linear decision boundary:

$$\hat{y} = \text{sign}(w^\top x + b)$$

where:

- $w \in \mathbb{R}^d$ is the *weight vector*
- $b \in \mathbb{R}$ is the *bias* (also called the intercept)
- $\text{sign}(z) = +1$ if $z \geq 0$, else -1

Geometric interpretation: The weight vector w defines a hyperplane $\{x : w^\top x + b = 0\}$ that separates the two classes. Points on one side are classified as $+1$, points on the other as -1 . The vector w is perpendicular to this hyperplane, and b controls its offset from the origin.

Update rule: Iterate through the training data. For each misclassified example (x_i, y_i) where $y_i \neq \hat{y}_i$:

$$w \leftarrow w + \eta \cdot y_i \cdot x_i$$

$$b \leftarrow b + \eta \cdot y_i$$

where $\eta > 0$ is the learning rate.

Why this update works: When a point is misclassified, the update “nudges” the decision boundary toward correctly classifying that point:

- If $y_i = +1$ but we predicted -1 , then $w^\top x_i + b < 0$ (too negative). Adding ηx_i to w increases $w^\top x_i$, pushing it toward positive territory.
- If $y_i = -1$ but we predicted $+1$, then $w^\top x_i + b > 0$ (too positive). Subtracting ηx_i from w decreases $w^\top x_i$, pushing it toward negative territory.

Perceptron Convergence Theorem: If the training data is *linearly separable* (there exists a hyperplane perfectly separating positive from negative examples), the perceptron algorithm converges in a finite number of steps. Specifically, if all points satisfy $\|x_i\| \leq R$ and the best separating hyperplane has margin γ (minimum distance from any point to the boundary), then the number of mistakes is bounded by $(R/\gamma)^2$.

Intuition: Each update rotates the weight vector toward correctly classifying the current mistake. If a perfect separator exists, we eventually find it. The bound shows that wider margins (easier problems) lead to faster convergence.

The perceptron generated enormous excitement. The New York Times ran the headline: “New Navy Device Learns By Doing; Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser.” Funding flowed in from the US Navy, and neural networks seemed poised to deliver on AI’s promises. Rosenblatt made bold claims about perceptrons eventually being able to “walk, talk, see, write, reproduce itself and be conscious of its existence.”

5 The First AI Winter (1970s)

Section Summary

Minsky and Papert's *Perceptrons* (1969) proved that single-layer networks cannot learn non-linearly-separable functions like XOR. Combined with overpromising and the Lighthill Report, this triggered the first AI funding collapse. The deeper lesson: limitations of a specific model do not doom an entire paradigm, but the distinction was lost in the backlash.

The optimism was short-lived. In 1969, Marvin Minsky and Seymour Papert published *Perceptrons*, a mathematical analysis that proved devastating limitations of single-layer perceptrons.

NB!

[The XOR Problem] Single-layer perceptrons cannot learn functions that are not **linearly separable**. The canonical example is XOR (exclusive or):

x_1	x_2	$x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

No single hyperplane (line in 2D) can separate the positive examples $\{(0, 1), (1, 0)\}$ from the negative examples $\{(0, 0), (1, 1)\}$ —they lie on opposite corners of a square.

This is not merely an artificial counterexample. Many real-world classification problems are not linearly separable, making single-layer perceptrons inadequate for practical applications.

The XOR Problem: Solutions and Legacy

The XOR problem admits several solutions, each historically significant:

1. **Multi-layer networks**: A hidden layer can learn intermediate representations. XOR can be decomposed as $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$, which breaks into two linearly separable sub-problems that a hidden layer can compute. This was known in 1969, but no efficient algorithm for training such networks was available.
2. **Feature engineering**: Map inputs to a higher-dimensional space where they become linearly separable. For XOR, adding the feature $x_1 \cdot x_2$ suffices:

x_1	x_2	x_1x_2	y
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Now the hyperplane $x_1 + x_2 - 2x_1x_2 = 0.5$ separates the classes. The key insight: the right representation makes the problem easy.

3. **The kernel trick** (developed later for SVMs): Implicitly map to high-dimensional feature spaces via a kernel function $k(x, x') = \langle \phi(x), \phi(x') \rangle$. We need only compute inner products in feature space, not the features themselves.

Connection to modern deep learning: Deep networks learn hierarchical feature representations automatically—the hidden layers construct the “features” that make the problem linearly separable in the final layer. This is precisely the representational power that Minsky and Papert noted was missing from single-layer networks.

While multi-layer networks could in principle overcome these limitations (a two-layer network can easily compute XOR), no efficient algorithm for training them was known at the time. The perceptron learning rule only works for single layers—it provides no guidance for adjusting weights in hidden layers.

The impact of *Perceptrons* was amplified by broader problems in AI:

- **Machine translation failures**: Early optimism about automatic translation proved unfounded; the ALPAC report (1966) recommended cutting funding
- **The Lighthill Report** (1973): A British government report highly critical of AI progress, leading to funding cuts in the UK
- **Unrealistic expectations**: Early predictions had set expectations impossibly high
- **Hardware limitations**: Computers of the 1970s were vastly underpowered for the ambitions of AI researchers
- **Scaling failures**: Techniques that worked on toy problems failed to scale to real-world complexity

Funding collapsed dramatically. DARPA cut AI funding; university programmes shrank. This period (roughly 1974–1980) became known as the **First AI Winter**.

Lessons from the First AI Winter

- Overpromising and underdelivering damages a field's credibility for years
- Proving limitations of one model (single-layer perceptrons) doesn't doom the entire approach (neural networks)
- Hardware constraints were severe—many ideas were ahead of their time computationally
- The gap between toy problems and real-world applications was systematically underestimated
- Theoretical results (like the XOR limitation) can have outsized impact on funding and perception

6 Knowledge-Based Systems and Expert Systems (1980s)

Section Summary

Expert systems dominated 1980s AI: encode human expertise as explicit rules, then reason over them. Commercial successes like R1/XCON showed real value. But fundamental problems emerged—the knowledge acquisition bottleneck (hard to extract and encode expertise), brittleness (failure outside narrow domains), and maintenance burden. These limitations would later motivate the shift toward learning from data.

AI revival came through a different paradigm: **knowledge-based systems**. Rather than learning from data, these systems encoded human expertise directly. The core philosophy was articulated as: “Don’t tell the program what to do, tell it what to know.”

A knowledge-based system has two components:

1. **Knowledge base:** Facts and rules about a domain, typically encoded in logic or rule-based formalisms (IF-THEN rules, semantic networks, frames)
2. **Inference engine:** Mechanisms for deriving new conclusions from the knowledge base (forward chaining, backward chaining, resolution)

Expert systems were knowledge-based systems designed to emulate human experts in narrow, well-defined domains. The key insight was that much human expertise could be captured as heuristic rules, even if the underlying theory was incomplete.

Notable examples:

- **MYCIN** (Stanford, 1970s): Diagnosed bacterial infections and recommended antibiotics. Used *certainty factors* to handle uncertainty. Performed comparably to human experts in blind tests, but was never deployed clinically due to liability concerns.
- **R1/XCON** (DEC, 1980s): Configured VAX computer systems. Saved DEC an estimated \$40 million annually by reducing errors and expert time. One of the few commercially successful expert systems.
- **DENDRAL** (Stanford, 1960s–1970s): Identified molecular structures from mass spectrometry data. A pioneering application of AI to scientific discovery.

Rule-Based Inference

Expert systems typically used production rules of the form:

IF ⟨condition⟩ THEN ⟨action⟩

For example, in MYCIN:

```
IF the infection is primary-bacteremia
AND the site of the culture is a sterile site
AND the suspected portal of entry is the GI tract
THEN there is suggestive evidence (0.7) that
    the identity of the organism is Bacteroides
```

Inference directions:

- **Forward chaining** (data-driven): Start from known facts, apply rules to derive new facts, continue until the goal is reached or no more rules apply. “Given what I know, what can I conclude?”
- **Backward chaining** (goal-driven): Start from the goal, find rules whose conclusions match, then try to establish those rules’ premises (recursively). “To prove this goal, what do I need to establish?”

This is equivalent to resolution theorem proving in propositional or first-order logic, giving expert systems a firm logical foundation.

The expert systems boom of the early 1980s created a commercial AI industry. Companies like Teknowledge, IntelliCorp, and Symbolics sold both expert system shells (tools for building systems) and specialised AI hardware (Lisp machines). Japan launched the ambitious Fifth Generation Computer Project (1982–1992) aiming to build “intelligent computers.”

However, fundamental problems emerged:

- **Knowledge acquisition bottleneck:** Extracting expertise from human experts is difficult, time-consuming, and expensive. Experts often cannot articulate their knowledge explicitly—they “just know.”
- **Brittleness:** Systems failed unpredictably when encountering situations outside their encoded knowledge. They had no common sense to fall back on.
- **Maintenance nightmare:** Knowledge bases became unwieldy and hard to update. Adding new rules could have unexpected interactions with existing rules.
- **No learning:** Systems could not improve from experience; all knowledge had to be hand-coded.

7 The Connectionist Revival (1980s)

Section Summary

Backpropagation (popularised 1986) enabled training multi-layer networks by efficiently computing gradients via the chain rule. This solved the credit assignment problem: determining which weights to blame for errors. The PDP books provided a theoretical framework arguing that intelligence emerges from distributed representations in parallel networks. Key figures: Rumelhart, Hinton, Williams, McClelland.

While expert systems dominated commercial AI, neural networks experienced a quiet renaissance in academic research. The key breakthrough was **backpropagation**—an efficient algorithm for training multi-layer networks by propagating error signals backwards through the network. Though backpropagation was discovered independently by several researchers (Paul Werbos in his 1974 PhD thesis, David Parker in 1985, Yann LeCun in 1985), the 1986 paper by Rumelhart, Hinton, and Williams in *Nature* brought it to widespread attention. Titled “Learning representations by back-propagating errors,” it demonstrated that:

1. Multi-layer networks could learn useful internal representations automatically
2. These representations captured meaningful structure in the data
3. XOR and similar non-linearly-separable problems were easily solved
4. The representations learned were often interpretable and interesting

Backpropagation: Core Idea

For a feedforward network computing a function $f_\theta(x)$ with parameters θ (all the weights), and a loss function $L(f_\theta(x), y)$ measuring the error on example (x, y) , we need the gradient $\frac{\partial L}{\partial \theta}$ to perform gradient descent.

The chain rule gives, for a weight $w_{ij}^{(l)}$ connecting unit i in layer $l - 1$ to unit j in layer l :

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \cdot \frac{\partial a_j^{(l)}}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} \cdot h_i^{(l-1)}$$

Unpacking the notation:

- $a_j^{(l)} = \sum_i w_{ij}^{(l)} h_i^{(l-1)}$ is the *pre-activation* at unit j in layer l
- $h_i^{(l-1)}$ is the *activation* (output) of unit i in the previous layer
- $\delta_j^{(l)} = \frac{\partial L}{\partial a_j^{(l)}}$ is the “error signal” at unit j —how much the loss would change if this pre-activation changed

The key insight: $\delta_j^{(l)}$ can be computed recursively from layer $l + 1$:

$$\delta_j^{(l)} = \sigma'(a_j^{(l)}) \sum_k w_{jk}^{(l+1)} \delta_k^{(l+1)}$$

where σ' is the derivative of the activation function. Errors “back-propagate” from output to input: we compute δ at the output layer (from the loss), then propagate backwards through the network.

This reduces the complexity from $O(W^2)$ (naive numerical differentiation, perturbing each weight and measuring the effect) to $O(W)$ where W is the total number of weights—a massive speedup that makes training practical.

The Credit Assignment Problem: In a multi-layer network, if the output is wrong, which weights are responsible? Backpropagation provides an answer: each weight’s “blame” is proportional to its gradient. Weights that could have reduced the loss receive larger gradients.

Connection to modern ML: Backpropagation remains the foundation of deep learning. Modern frameworks (PyTorch, JAX) implement *automatic differentiation*—a generalisation that computes gradients through arbitrary computation graphs, not just feedforward networks. The principle is the same: decompose complex functions into elementary operations, then chain-rule backwards.

The **Parallel Distributed Processing** (PDP) volumes (1986), edited by Rumelhart and McClelland, provided a comprehensive framework for connectionist cognitive science. The “PDP research group” at UCSD argued that intelligence emerges from many simple, neuron-like units operating in parallel, with knowledge stored in the connection weights rather than in explicit rules. This offered a radically different vision of mind than symbolic AI.

8 The Second AI Winter (Late 1980s–Early 1990s)

Section Summary

The expert systems bubble burst, and neural networks hit practical walls (vanishing gradients, computational cost). The term “AI” became toxic; researchers rebranded as “machine learning” or “computational intelligence.” Yet foundational work continued: the theoretical framework for statistical learning emerged, and key algorithms (SVMs, boosting) were developed. Winters end.

Despite academic progress in neural networks, a second AI winter set in around 1987–1993:

- **Expert systems bust:** The promised benefits of expert systems failed to materialise at scale. Many expensive projects were abandoned.
- **Hardware disruption:** Specialised AI hardware (Lisp machines from Symbolics, LMI, TI) became obsolete almost overnight. The rise of cheap, powerful workstations from Sun and the personal computer revolution—driven by IBM and Apple—eliminated the market for expensive specialised machines. General-purpose hardware caught up and surpassed dedicated AI machines in cost-effectiveness.
- **Japan’s Fifth Generation failure:** The ambitious project failed to achieve its goals, dampening enthusiasm for AI investment.
- **Neural network limitations:**
 - **Vanishing gradients:** In deep networks, gradients became exponentially small in early layers, making them effectively untrainable
 - **Computational cost:** Training was slow on available hardware; networks were limited to a few layers
 - **Limited theory:** Why networks worked (or didn’t) was poorly understood; training felt like “alchemy”
- **Alternative paradigms:** Some researchers pursued “embodied” or “situated” AI (Brooks’ subsumption architecture), arguing that intelligence required physical grounding in the world, not just symbol manipulation. This fragmented the field and raised questions about whether traditional AI approaches were fundamentally misguided.

The term “AI” became commercially toxic. Researchers strategically rebranded their work as “machine learning,” “computational intelligence,” “knowledge discovery,” “data mining,” or “pattern recognition” to escape the stigma and maintain funding.

9 Unsupervised Learning

Section Summary

Unsupervised learning discovers structure without labels: clustering (grouping similar points), dimensionality reduction (finding compact representations), density estimation (modelling $p(x)$). Key developments: Kohonen’s self-organising maps, Hopfield networks for associative memory, Boltzmann machines for probabilistic modelling. These ideas resurface in modern generative models.

Alongside supervised learning, researchers developed methods for **unsupervised learning**—finding structure in data without labels. Where supervised learning asks “what is the correct answer for this input?”, unsupervised learning asks “what patterns exist in this data?”

Unsupervised Learning: Formal Definition

Given unlabelled data $\mathcal{D} = \{x_1, \dots, x_n\}$ where $x_i \in \mathcal{X}$, the goal is to discover structure in the data distribution $p(x)$. There is no “correct answer” to supervise learning; success is measured by whether the discovered structure is useful or meaningful.

Key distinction from supervised learning: There are no labels y_i . We only have inputs, and we seek patterns within them.

Common tasks include:

- **Clustering:** Partition data into groups of similar points (k -means, hierarchical clustering, DBSCAN). Each point is assigned to a cluster $z_i \in \{1, \dots, K\}$.
- **Dimensionality reduction:** Find low-dimensional representations that preserve important structure (PCA, autoencoders, t-SNE, UMAP). Map $x \in \mathbb{R}^d$ to $z \in \mathbb{R}^k$ where $k \ll d$.
- **Density estimation:** Model the probability distribution $p(x)$ explicitly (Gaussian mixture models, kernel density estimation).
- **Generative modelling:** Learn to sample new data from the learned distribution (VAEs, GANs, diffusion models).

Why is this useful?

- Labels are expensive to obtain; unlabelled data is abundant
- Understanding data structure helps with downstream tasks
- Generative models enable data augmentation, anomaly detection, and creative applications
- Learned representations often transfer to supervised tasks

Key developments in unsupervised learning during this period included:

- **Self-Organising Maps** (Kohonen, 1982): Neural networks that learn topological maps of high-dimensional data, preserving neighbourhood relationships. A form of competitive learning where nearby units respond to similar inputs.
- **Hopfield Networks** (1982): Recurrent networks that function as associative (content-addressable) memory, using an energy function framework from statistical physics. The network stores patterns as local minima of an energy landscape; retrieval involves gradient descent to the nearest minimum.
- **Boltzmann Machines** (Hinton & Sejnowski, 1983): Stochastic neural networks that learn probability distributions over their inputs, providing a principled probabilistic framework for neural computation. The connection to statistical mechanics (Boltzmann distributions) gave theoretical grounding.

10 Reinforcement Learning

Section Summary

Reinforcement learning addresses sequential decision-making: an agent takes actions in an environment, receives rewards, and learns to maximise cumulative reward. Formalised via Markov Decision Processes. Key algorithms: temporal difference learning (Sutton), Q-learning (Watkins). TD-Gammon (1992) demonstrated that RL could achieve expert-level performance through self-play.

A third paradigm emerged: **reinforcement learning** (RL)—learning from interaction with an environment through trial and error. Unlike supervised learning (which requires labelled examples) and unsupervised learning (which has no feedback), RL learns from *rewards* that signal how good an action was, without being told the correct action directly.

Reinforcement Learning: Formal Framework

The standard framework is the **Markov Decision Process** (MDP), consisting of:

- \mathcal{S} : Set of states the environment can be in
- \mathcal{A} : Set of actions the agent can take
- $P(s'|s, a)$: Transition probability—the probability of reaching state s' after taking action a in state s
- $R(s, a, s')$: Reward function—the immediate reward received for the transition
- $\gamma \in [0, 1)$: Discount factor—how much to value future rewards relative to immediate ones

Unpacking the discount factor: γ close to 0 means the agent is “myopic”—it cares mostly about immediate rewards. γ close to 1 means it values long-term rewards nearly as much as immediate ones. The constraint $\gamma < 1$ ensures the sum of discounted rewards is finite.

The goal is to learn a **policy** $\pi : \mathcal{S} \rightarrow \mathcal{A}$ (or $\pi(a|s)$ for stochastic policies) that maximises expected cumulative discounted reward. The **value function** measures expected future reward from a state under policy π :

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \mid s_0 = s \right]$$

Reading this equation: Starting from state s , following policy π , what is the expected sum of discounted future rewards? The expectation is over the stochasticity of both the policy and the environment transitions.

The **Bellman equation** characterises the optimal value function:

$$V^*(s) = \max_a \left[R(s, a) + \gamma \sum_{s'} P(s'|s, a) V^*(s') \right]$$

Interpreting the Bellman equation: The optimal value of state s equals the value of the best action, which is the immediate reward plus the discounted expected value of the next state. This recursive equation is the foundation of dynamic programming algorithms (value iteration, policy iteration) that solve MDPs when the model is known.

Key developments in RL:

- **Temporal Difference Learning** (Sutton, 1988): Learning value functions from experience without waiting for episode completion. TD learning *bootsraps*—it updates estimates based on other estimates, rather than waiting for ground truth.
- **Q-Learning** (Watkins, 1989): A model-free algorithm for learning optimal policies without knowing transition probabilities. Learns the Q-function $Q(s, a) = R(s, a) + \gamma \sum_{s'} P(s'|s, a) V^*(s')$ —the value of taking action a in state s and then acting optimally.
- **TD-Gammon** (Tesauro, 1992): An RL system that achieved expert-level backgammon play through self-play, learning entirely from game outcomes. A landmark demonstration

of RL's potential—the system discovered novel strategies that surprised human experts.

11 The Statistical ML Renaissance (1990s–2000s)

Section Summary

Machine learning matured into a rigorous discipline, importing tools from statistics: PAC learning theory, VC dimension, cross-validation, probabilistic graphical models. SVMs combined strong theory with excellent practice. Ensemble methods (boosting, random forests) dominated competitions. The focus shifted from “can we build intelligent machines?” to “can we learn accurate predictors with provable guarantees?”

The 1990s saw machine learning mature into a rigorous scientific discipline, borrowing heavily from statistics and developing its own theoretical foundations. This period established ML as a respectable academic field with principled methodology.

Key Themes of the Statistical ML Era

- **Rigour:** Formal learning theory, generalisation bounds, PAC (Probably Approximately Correct) learning, VC dimension
- **Probabilistic methods:** Bayesian inference, graphical models, principled uncertainty quantification
- **Evaluation discipline:** Cross-validation, held-out test sets, standardised benchmarks, statistical significance testing
- **Reproducibility:** Open datasets (UCI repository), shared code, detailed experimental protocols

11.1 Support Vector Machines

Vladimir Vapnik and colleagues developed **Support Vector Machines** (SVMs) at Bell Labs, which dominated practical ML from the mid-1990s through the 2000s. SVMs combined several appealing properties:

- **Strong theoretical foundations:** Grounded in VC theory and structural risk minimisation, providing guarantees about generalisation
- **The “kernel trick”:** Implicitly computing in high-dimensional feature spaces without explicitly constructing them, enabling nonlinear classification with linear algorithms
- **Convex optimisation:** The training problem is a quadratic program with a unique global optimum—no local minima to worry about
- **Excellent empirical performance:** State-of-the-art results on many benchmarks
- **Sparse solutions:** Only a subset of training points (support vectors) matter, making prediction efficient

The Kernel Trick: Key Insight

The kernel trick addresses the XOR problem elegantly. Recall that non-linearly-separable data can become separable in a higher-dimensional feature space $\phi(x)$. But computing in high dimensions is expensive—potentially intractable if the feature space is infinite-dimensional.

The insight: Many algorithms (including SVMs and perceptrons) depend on data only through inner products $\langle x_i, x_j \rangle$. If we can compute $k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$ directly—without explicitly computing ϕ —we get the representational power of high-dimensional features at the cost of computing pairwise similarities.

Example kernels:

- **Polynomial kernel:** $k(x, x') = (1 + x^\top x')^d$ implicitly maps to a space containing all monomials up to degree d
- **RBF (Gaussian) kernel:** $k(x, x') = \exp(-\|x - x'\|^2/2\sigma^2)$ corresponds to an infinite-dimensional feature space

Connection to deep learning: Deep networks learn feature representations $\phi(x)$ end-to-end from data, rather than choosing them a priori via a kernel. The relationship between kernel methods and neural networks remains an active research area (neural tangent kernels, infinite-width limits of neural networks).

11.2 Ensemble Methods

The 1990s also saw the rise of ensemble methods—combining multiple models to improve predictions:

- **Bagging** (Breiman, 1996): Train multiple models on bootstrap samples (sampling with replacement), average their predictions. Reduces variance.
- **Boosting** (Freund & Schapire, 1997): Sequentially train weak learners, with each focusing on examples the previous ones got wrong. AdaBoost won the Gödel Prize for its theoretical elegance.
- **Random Forests** (Breiman, 2001): Bagging applied to decision trees, plus random feature subsets at each split. Remarkably effective and robust.
- **Stacking**: Use one model’s predictions as input features for another model.

11.3 Probabilistic Graphical Models

Judea Pearl’s work on Bayesian networks and causal inference brought principled probabilistic reasoning into AI. His 1988 book *Probabilistic Reasoning in Intelligent Systems* was transformative. Graphical models provided:

- **Principled uncertainty quantification:** Representing and computing with probability distributions
- **Modular representation:** Factorising complex joint distributions according to conditional independence
- **Efficient inference algorithms:** Belief propagation, variable elimination, MCMC sampling

- **Causal reasoning:** Pearl’s later work on causality (the “do-calculus”) distinguished correlation from causation

12 The Deep Learning Revolution (2010s–Present)

Section Summary

Deep learning resurgence began with Hinton’s 2006 work on pretraining, then exploded with AlexNet’s 2012 ImageNet victory. Key enablers: GPU compute, massive datasets, architectural innovations (ReLU, dropout, residual connections). Transformers (2017) revolutionised NLP, leading to large language models. The field now grapples with scale, capabilities, and limitations of these systems.

The current era began around 2006–2012 with dramatic breakthroughs that revived neural networks under the banner of “deep learning.”

Key Timeline: Deep Learning Era

- **2006:** Hinton’s deep belief networks—greedy layer-wise pretraining enables training of deep networks
- **2009:** GPU training demonstrated (Raina et al.)—order of magnitude speedup
- **2012:** AlexNet wins ImageNet by large margin (15.3% vs 26.2% error)—CNNs take over computer vision
- **2014:** GANs introduced (Goodfellow et al.)—generative modelling revolution
- **2014:** Sequence-to-sequence models with attention (Bahdanau et al.)
- **2015:** ResNet enables training of very deep networks (152 layers) via skip connections
- **2016:** AlphaGo defeats world Go champion Lee Sedol
- **2017:** “Attention Is All You Need”—Transformers introduced, eliminating recurrence
- **2018:** BERT—bidirectional pretraining for NLP
- **2020:** GPT-3 (175B parameters)—emergence of in-context learning
- **2022:** ChatGPT brings LLMs to mainstream attention; diffusion models for image generation
- **2023–:** Multimodal models (GPT-4V, Gemini), reasoning capabilities, agent systems

12.1 Why Now? The Convergence of Factors

Several factors converged to enable the deep learning revolution:

1. **Compute:** GPUs provided massive parallelism for matrix operations. NVIDIA’s CUDA (2007) made GPU programming accessible. Training that would take months on CPUs took days on GPUs. A modern GPU can perform 10^{13} floating-point operations per second—orders of magnitude more than 1980s hardware.

2. **Data:** The internet generated unprecedented quantities of labelled and unlabelled data. ImageNet (14M images, 1000 classes) provided a challenging benchmark. Social media, digitised text, and web scraping created training sets at previously impossible scales. Web text provides trillions of tokens for language model training.
3. **Algorithms:** Better architectures and training techniques solved old problems:
 - **ReLU activations:** $\max(0, x)$ instead of sigmoid, reducing vanishing gradients
 - **Dropout:** Randomly zeroing activations during training, providing effective regularisation
 - **Batch normalisation:** Normalising layer inputs, stabilising training
 - **Residual connections:** $f(x) + x$ instead of $f(x)$, enabling very deep networks
 - **Adam** and other adaptive optimisers improved convergence
4. **Software infrastructure:** Frameworks like Theano, TensorFlow, and PyTorch made experimentation accessible. Automatic differentiation eliminated manual gradient derivation.
5. **Frictionless reproducibility:** Preprints on arXiv, open-source code, and rapid iteration accelerated progress. Downloading code and pretrained models to replicate results became routine.
6. **Industry investment:** Tech companies (Google, Facebook, Microsoft, etc.) invested billions in AI research, attracting talent and resources.

12.2 Convolutional Networks and the Path to ImageNet

The 2012 breakthrough built on decades of work. **Yann LeCun** and colleagues developed **convolutional neural networks** (CNNs) in the late 1980s, demonstrating their effectiveness on handwritten digit recognition (LeNet). The key insight: exploit the spatial structure of images through local connectivity and weight sharing.

Convolutional Neural Networks: Core Ideas

CNNs incorporate inductive biases suited to images:

- **Local connectivity:** Each neuron connects only to a small spatial region (receptive field), not the entire image. This assumes nearby pixels are more relevant than distant ones.
- **Weight sharing:** The same filter (set of weights) is applied across all spatial positions. If a pattern is useful to detect in one location, it's useful everywhere.
- **Pooling:** Downsampling reduces spatial resolution, providing translation invariance and reducing computation.

Why these constraints help: They dramatically reduce the number of parameters (compared to fully-connected networks) while encoding prior knowledge that images have spatial structure. A fully-connected layer for a $224 \times 224 \times 3$ image would have $\sim 150,000$ input units; a convolutional layer with 64 3×3 filters has only $64 \times 3 \times 3 \approx 1,700$ parameters.

Historical note: LeCun's LeNet (1989) was deployed commercially for cheque reading in the 1990s—a genuine industrial success during the “AI winter.” The architecture ideas were proven; what was missing was sufficient compute and data for harder problems.

AlexNet (2012) applied these principles at scale: 8 layers, 60 million parameters, trained on 1.2 million ImageNet images using GPUs. Its dramatic victory (reducing error from 26% to 15%) signalled that deep learning worked on real problems.

12.3 Transformers and Language Models

The Transformer architecture (Vaswani et al., 2017, “Attention Is All You Need”) revolutionised sequence modelling. Previous approaches (RNNs, LSTMs) processed sequences step-by-step, creating bottlenecks for long sequences. Transformers process all positions in parallel through the **attention mechanism**—allowing models to dynamically focus on relevant parts of the input regardless of distance.

Scaled Dot-Product Attention

The core operation of Transformers. Given:

- Queries $Q \in \mathbb{R}^{n \times d_k}$ (what we're looking for)
- Keys $K \in \mathbb{R}^{m \times d_k}$ (what we're matching against)
- Values $V \in \mathbb{R}^{m \times d_v}$ (what we're retrieving)

Attention computes:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

Step-by-step interpretation:

1. QK^\top computes similarity scores between each query and all keys. Entry (i, j) measures how much query i “matches” key j .
2. Dividing by $\sqrt{d_k}$ prevents scores from becoming too large, which would make softmax saturate (outputting nearly one-hot distributions with vanishing gradients).
3. The softmax converts scores to attention weights (probabilities that sum to 1 across keys).
4. Finally, we take a weighted sum of values—each query retrieves a combination of values, weighted by attention.

Intuition: Think of attention as a soft lookup table. The query asks “what information do I need?”, the keys say “here’s what I have”, and the values are the actual information retrieved. Unlike a hard lookup (which retrieves exactly one entry), attention retrieves a weighted blend.

Why $\sqrt{d_k}$? The dot product of two random vectors with entries of variance 1 has variance d_k . Scaling by $\sqrt{d_k}$ restores unit variance, keeping gradients well-behaved.

Multi-head attention runs several attention operations in parallel with different learned projections, allowing the model to attend to different types of relationships simultaneously (e.g., syntactic vs semantic relationships in language).

Key innovation over RNNs: Unlike RNNs that process tokens sequentially, transformers process all positions in parallel. Long-range dependencies become $O(1)$ in path length rather than $O(n)$. This parallelism enables efficient training on modern hardware.

Large Language Models (LLMs) trained on internet-scale text have demonstrated remarkable capabilities:

- **In-context learning:** Learning from examples provided in the prompt, without updating weights
- **Chain-of-thought reasoning:** Generating step-by-step reasoning that improves accuracy on complex tasks
- **Code generation:** Writing, explaining, and debugging code across many languages

- **Multi-step problem solving:** Breaking down complex tasks and executing them
- **Instruction following:** Generalising to novel tasks described in natural language

The **scaling hypothesis** suggests that many capabilities emerge from simply training larger models on more data—capabilities that were absent in smaller models appear suddenly at scale (“emergence”). This has driven an arms race in model size, from millions to billions to trillions of parameters.

NB!

[Current Limitations] The capabilities of current AI systems, while impressive, have significant limitations:

- **Hallucination:** Confident generation of plausible-sounding but false information
- **Lack of grounding:** Models manipulate symbols without understanding their real-world referents
- **Brittleness to distribution shift:** Performance degrades on data unlike the training distribution
- **No robust common-sense reasoning:** Failures on simple physical or social reasoning that children handle easily
- **Unclear generalisation:** What models have “learned” versus “memorised” remains debated
- **Alignment challenges:** Ensuring models behave as intended and avoid harmful outputs
- **Environmental cost:** Training large models requires enormous energy

The gap between benchmark performance and real-world robustness remains significant. Current systems are “narrow AI”—highly capable in specific domains but far from human-like general intelligence.

13 Summary: Recurring Themes in AI History

Section Summary

AI’s history reveals consistent patterns: hype cycles, the symbolic-connectionist tension, hardware as a bottleneck, the importance of evaluation, and productive cross-pollination between fields. Understanding these patterns provides perspective on current advances and appropriate scepticism about future predictions.

Patterns Across AI History

1. **Hype cycles:** Periods of optimism and investment followed by “winters” when expectations aren’t met. We may be in a hype period now.
2. **Symbolic vs. connectionist:** The recurring tension between rule-based (explicit knowledge) and learning-based (implicit knowledge) approaches. Currently, connectionism dominates, but hybrid approaches are emerging.
3. **Hardware constraints:** Progress often waits for computational capacity. Many ideas from the 1980s only became practical with modern GPUs.
4. **The importance of data:** Modern ML success is as much about data quantity and quality as algorithmic innovation. The “unreasonable effectiveness of data.”
5. **Evaluation matters:** Rigorous benchmarks drive progress but can be gamed or become saturated. What gets measured gets optimised.
6. **Transfer between fields:** Ideas from statistics (Bayesian inference), physics (energy-based models, diffusion), neuroscience (attention, sparse coding), and linguistics have all contributed.
7. **The bitter lesson** (Rich Sutton): Methods that leverage computation scale better than methods that leverage human knowledge. General methods + more compute beats clever, domain-specific methods.

The history of AI is a story of grand ambitions, humbling setbacks, and gradual progress. Understanding this history provides perspective on current advances and appropriate caution about future predictions. Every generation of AI researchers has believed they were on the verge of a breakthrough; sometimes they were right (deep learning), often they were premature (expert systems, early neural networks).

The fundamental questions remain open: What is intelligence? Can machines truly understand, or only simulate understanding? Will current approaches scale to general AI, or will fundamentally new ideas be needed? History suggests humility about predictions—but also persistence. Ideas dismissed as failures often return, transformed, when conditions are right.

13.1 Looking Ahead

The concepts introduced here—supervised learning, unsupervised learning, reinforcement learning, neural networks, loss functions, optimisation—will be developed rigorously in subsequent weeks. The historical framing should help answer the recurring question: *why this approach?* Modern techniques are not arbitrary; they emerged from decades of theoretical development and empirical discovery.

A note on perspective: we are currently in a period of intense progress and enthusiasm. History suggests caution about extrapolating current trends indefinitely. The interesting questions are not “will AI keep improving?” but rather “what are the fundamental limitations?” and “what new ideas might be needed?” These questions will recur throughout the course.

ML Lecture Notes: Week 2

Training, Divergence, Loss & Optimisation

14 The Optimisation Framework

Section Summary

Machine learning casts model fitting as optimisation: find parameters $\hat{\theta} = \arg \min_{\theta} \mathcal{L}(\theta)$. The loss function \mathcal{L} encodes what “fitting well” means. This section introduces empirical risk minimisation as the unifying framework.

Machine learning is fundamentally about finding parameters that make models fit data well. This week, we formalise what “fitting well” means and develop the mathematical machinery to achieve it. The central insight is that *modelling is optimisation*: we define a measure of fit (the loss function), then search for parameters that optimise it.

Parameter Estimation as Optimisation

$$\hat{\theta} = \arg \min_{\theta} \mathcal{L}(\theta)$$

where:

- $\hat{\theta}$ is the **point estimate**—our best guess for the parameters
- $\mathcal{L}(\theta)$ is the **loss function**—measures how poorly the model fits
- $\arg \min$ returns the parameter value that minimises the loss

Unpacking the notation: The symbol $\arg \min$ is distinct from \min . While $\min_{\theta} \mathcal{L}(\theta)$ returns the *value* of the minimum (how small the loss gets), $\arg \min_{\theta} \mathcal{L}(\theta)$ returns the *argument* that achieves that minimum (which parameter values make the loss smallest). For example, if $\mathcal{L}(\theta) = (\theta - 3)^2$, then $\min_{\theta} \mathcal{L}(\theta) = 0$ but $\arg \min_{\theta} \mathcal{L}(\theta) = 3$.

The choice of loss function \mathcal{L} determines what “fitting well” means. Different losses encode different priorities: accuracy, robustness, calibration, or interpretability. This is a *design choice*—there is no single “correct” loss function, and the choice should reflect what we actually care about in a given application.

Key Insight

Modelling is empirical risk minimisation. We define risk through a loss function and optimise to minimise it. The loss function is a design choice that should reflect what we care about.

15 Maximum Likelihood Estimation (MLE)

Section Summary

MLE chooses parameters to maximise $p(\mathcal{D}|\theta)$. Under i.i.d. assumptions, this factorises as a product, which we convert to a sum via logarithms (the NLL). Key result: MLE is equivalent to minimising KL divergence from the true distribution to the model.

Maximum Likelihood Estimation is a principled approach that chooses parameters to maximise the probability of observing the data we actually observed. Rather than inventing an ad-hoc loss function, MLE derives the loss from probabilistic principles.

Maximum Likelihood Estimation

Given data $\mathcal{D} = \{(x_1, y_1), \dots, (x_n, y_n)\}$ and a probabilistic model $p(y|x, \theta)$:

$$\hat{\theta}_{\text{MLE}} = \arg \max_{\theta} p(\mathcal{D}|\theta) = \arg \max_{\theta} \prod_{i=1}^n p(y_i|x_i, \theta)$$

The product assumes **i.i.d.** (independent and identically distributed) observations.

Unpacking the MLE formula: Let's break down what each part means:

- $p(\mathcal{D}|\theta)$ is the **likelihood**—the probability of observing our entire dataset \mathcal{D} , assuming the parameters are θ . Note the perspective: we treat the data as fixed (it's what we observed) and ask which parameter values would have made this data most probable.
- $\prod_{i=1}^n p(y_i|x_i, \theta)$ expresses the likelihood as a product over individual observations. This factorisation is only valid under the i.i.d. assumption.
- $\arg \max_{\theta}$ finds the parameter value that makes this probability largest.

This is the **frequentist** perspective—parameters are fixed but unknown constants, and data is the random variable. We're asking: “Given that the world works according to some fixed (unknown) parameters, what parameter values would make our observed data most likely?”

MLE as a Loss Function

MLE is just a method that chooses a particular loss function (the Negative Log-Likelihood). The key insight is that MLE seeks to find the model parameters that make the observed data maximally probable—equivalently, making the model as “close” as possible to the data in a probabilistic sense.

15.1 The i.i.d. Assumption

The factorisation $p(\mathcal{D}|\theta) = \prod_{i=1}^n p(y_i|x_i, \theta)$ relies on two assumptions:

1. **Independence:** Observations don't influence each other. Knowing y_1 tells us nothing about y_2 beyond what we already know from the model. Mathematically, this means $p(y_1, y_2|x_1, x_2, \theta) = p(y_1|x_1, \theta) \cdot p(y_2|x_2, \theta)$.
2. **Identical distribution:** All observations follow the same model with the same parameters. The function $p(y|x, \theta)$ is the same for all i —we're not using different models for different observations.

NB!

[The i.i.d. Assumption is Substantive] The i.i.d. assumption is **substantively meaningful** and often violated in practice. Examples of violations:

- **Time series data:** Today's stock price depends on yesterday's (temporal dependence)
- **Spatial data:** Nearby locations have correlated measurements (spatial correlation)
- **Clustered data:** Students within the same school are more similar (hierarchical structure)
- **Network data:** Connected individuals influence each other (relational dependence)

Violations can lead to underestimated standard errors and overconfident inference. When i.i.d. fails, we need more sophisticated models (time series models, mixed effects models, spatial models, etc.).

Why does violation matter? When observations are dependent, treating them as independent overcounts the effective information in the data. Imagine you survey 100 people, but they're all from the same family—you don't really have 100 independent data points about the general population. Standard errors computed under i.i.d. would be too small, making us overconfident in our estimates.

15.2 From Likelihood to Negative Log-Likelihood

Working with products is problematic for two reasons:

1. **Numerical instability:** Multiplying many small probabilities causes **underflow**—the computer rounds to zero. If each $p(y_i|x_i, \theta) \approx 0.01$, then with $n = 1000$ observations, the product is approximately 10^{-2000} , which is far smaller than the smallest number a computer can represent.
2. **Mathematical awkwardness:** Derivatives of products are messy (product rule applied repeatedly). For a product of n terms, the derivative has n terms, each involving a product of $n - 1$ factors.

Taking logarithms converts products to sums, solving both problems:

Negative Log-Likelihood (NLL)

$$\text{NLL}(\theta) = -\log p(\mathcal{D}|\theta) = -\sum_{i=1}^n \log p(y_i|x_i, \theta)$$

Since log is monotonically increasing:

$$\hat{\theta}_{\text{MLE}} = \arg \max_{\theta} p(\mathcal{D}|\theta) = \arg \min_{\theta} \text{NLL}(\theta)$$

Optimisers typically minimise, so we work with NLL rather than likelihood.

Why does this work? The logarithm is a *monotonically increasing* function: if $a > b$, then $\log(a) > \log(b)$. This means maximising $p(\mathcal{D}|\theta)$ is equivalent to maximising $\log p(\mathcal{D}|\theta)$. We

negate to convert maximisation to minimisation (since optimisation libraries typically minimise by default).

The key transformation uses the logarithm rule $\log(ab) = \log(a) + \log(b)$:

$$\log \prod_{i=1}^n p(y_i|x_i, \theta) = \sum_{i=1}^n \log p(y_i|x_i, \theta)$$

Why Negative Log-Likelihood?

1. **Numerical stability:** Sums don't underflow like products of small numbers. Log-probabilities are typically moderate negative numbers (e.g., -5) rather than tiny positive numbers (e.g., 0.007).
2. **Computational convenience:** Derivatives of sums are sums of derivatives: $\frac{d}{d\theta} \sum_i f_i(\theta) = \sum_i \frac{d}{d\theta} f_i(\theta)$.
3. **Convention:** Optimisation libraries minimise by default, so we negate.
4. **Interpretation:** NLL measures "surprise"—lower NLL means the data is less surprising under the model, indicating better fit. If the model assigns high probability to what we observed, $-\log p$ is small.

16 Linear Regression as MLE

Section Summary

Assuming Gaussian errors, MLE for regression reduces to minimising the sum of squared residuals. The analytic solution $\hat{\beta} = (X^\top X)^{-1} X^\top y$ has a beautiful geometric interpretation: $\hat{y} = X\hat{\beta}$ is the orthogonal projection of y onto the column space of X .

Linear regression emerges naturally from MLE when we assume normally distributed errors. This connection is profound: it tells us that the familiar least squares method isn't arbitrary—it's the principled thing to do when errors are Gaussian.

16.1 The Probabilistic Model

Assume each observation follows:

$$y_i \sim \mathcal{N}(\mu_i, \sigma^2) \quad \text{where} \quad \mu_i = x_i^\top \beta$$

This says: the response y_i is the linear prediction $x_i^\top \beta$ plus Gaussian noise with variance σ^2 .

Linear Regression Model

$$y_i = x_i^\top \beta + \epsilon_i \quad \text{where} \quad \epsilon_i \stackrel{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2)$$

Model parameters: $\theta = (\beta, \sigma^2)$

The probability density for observation y_i :

$$p(y_i|x_i, \theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - x_i^\top \beta)^2}{2\sigma^2}\right)$$

Unpacking the model components:

- $x_i^\top \beta = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \dots + \beta_p x_{ip}$ is the **linear predictor**—a weighted sum of features. Each β_j represents the contribution of feature j to the prediction.
- ϵ_i is the **error term**—captures everything the linear model doesn't explain. We assume it's random noise drawn from a Gaussian distribution.
- The notation $\epsilon_i \stackrel{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2)$ means: errors are independent across observations, each drawn from the same normal distribution with mean 0 and variance σ^2 .
- The Gaussian density formula gives the probability of observing y_i , which depends on how far y_i is from its predicted mean $x_i^\top \beta$, scaled by the noise level σ .

Linear Regression = Normal Model with Linear Predictor

Linear regression assumes:

1. **Normal Distribution:** Each observation y_i comes from a normal distribution
2. **Linear Relationship:** The mean $\mu_i = x_i^\top \beta$ is a linear combination of predictors
3. **Homoskedasticity:** The variance σ^2 is constant across all observations

These three assumptions together define the classical linear regression model.

16.2 Deriving the NLL

Taking the negative log of the Gaussian density for a single observation:

$$\begin{aligned} -\log p(y_i|x_i, \theta) &= -\log \left[\frac{1}{\sqrt{2\pi\sigma^2}} \exp \left(-\frac{(y_i - x_i^\top \beta)^2}{2\sigma^2} \right) \right] \\ &= -\log \frac{1}{\sqrt{2\pi\sigma^2}} - \log \exp \left(-\frac{(y_i - x_i^\top \beta)^2}{2\sigma^2} \right) \\ &= \frac{1}{2} \log(2\pi\sigma^2) + \frac{(y_i - x_i^\top \beta)^2}{2\sigma^2} \end{aligned}$$

Step-by-step breakdown:

1. We use $\log(ab) = \log(a) + \log(b)$ to separate the normalising constant from the exponential.
2. For the first term: $-\log(1/\sqrt{2\pi\sigma^2}) = -\log(2\pi\sigma^2)^{-1/2} = \frac{1}{2} \log(2\pi\sigma^2)$.
3. For the second term: $-\log(\exp(x)) = -x$ (since log and exp are inverses).

Summing over all observations:

$$\begin{aligned} \text{NLL}(\theta) &= -\sum_{i=1}^n \log p(y_i|x_i, \theta) \\ &= \sum_{i=1}^n \left[\frac{1}{2} \log(2\pi\sigma^2) + \frac{(y_i - x_i^\top \beta)^2}{2\sigma^2} \right] \\ &= \frac{n}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} \sum_{i=1}^n (y_i - x_i^\top \beta)^2 \end{aligned}$$

The MLE-RSS Connection

The NLL has two terms:

1. $\frac{n}{2} \log(2\pi\sigma^2)$: Depends only on σ^2 (constant w.r.t. β)
2. $\frac{1}{2\sigma^2} \sum_{i=1}^n (y_i - x_i^\top \beta)^2$: The sum of squared residuals, scaled by $1/(2\sigma^2)$

Since $\sigma^2 > 0$, minimising NLL over β is equivalent to minimising the sum of squared residuals. The Gaussian assumption justifies least squares!

MLE = Least Squares for Gaussian Errors

When we assume Gaussian errors, MLE for linear regression produces the same $\hat{\beta}$ as ordinary least squares (OLS). This means least squares isn't just a convenient computational method—it's the *statistically optimal* method under Gaussian assumptions.

This is why least squares is so ubiquitous: it's both computationally tractable (has a closed-form solution) and statistically principled (is the MLE under a reasonable model).

17 Residual Sum of Squares and the OLS Solution

Residual Sum of Squares

$$\text{RSS}(\beta) = \sum_{i=1}^n (y_i - x_i^\top \beta)^2 = \|y - X\beta\|_2^2 = (y - X\beta)^\top (y - X\beta)$$

where $X \in \mathbb{R}^{n \times p}$ is the design matrix and $y \in \mathbb{R}^n$ is the response vector.

Mean Squared Error: $\text{MSE}(\beta) = \frac{1}{n} \text{RSS}(\beta)$

Unpacking the notation: The RSS can be written in three equivalent ways:

1. **Summation form:** $\sum_{i=1}^n (y_i - x_i^\top \beta)^2$ sums the squared “residuals” (differences between observed and predicted values) over all observations.
2. **Norm form:** $\|y - X\beta\|_2^2$ is the squared Euclidean length of the residual vector. This emphasises that we're measuring the “distance” between y and $X\beta$ in n -dimensional space.
3. **Matrix form:** $(y - X\beta)^\top (y - X\beta)$ is the inner product of the residual vector with itself. This form is most useful for calculus.

The design matrix X has dimensions $n \times p$: rows are observations, columns are features. The i -th row of X is x_i^\top , so $(X\beta)_i = x_i^\top \beta$ is the predicted value for observation i .

17.1 The Analytic Solution

To find the minimum, we take the gradient and set it to zero. This is possible because RSS is a **convex quadratic** function of β —it has a unique global minimum (assuming X has full column rank).

Derivation of OLS Estimator

Step 1: Expand RSS using matrix algebra

$$\begin{aligned}\text{RSS}(\beta) &= (y - X\beta)^T(y - X\beta) \\ &= y^T y - y^T X \beta - \beta^T X^T y + \beta^T X^T X \beta \\ &= y^T y - 2\beta^T X^T y + \beta^T X^T X \beta\end{aligned}$$

(The middle terms are equal since they're scalars: $(y^T X \beta)^T = \beta^T X^T y$, and a scalar equals its transpose.)

Step 2: Take the gradient w.r.t. β

Using matrix calculus identities:

- $\nabla_{\beta}(\beta^T a) = a$ for any vector a
- $\nabla_{\beta}(\beta^T A \beta) = 2A\beta$ for any symmetric matrix A

$$\nabla_{\beta} \text{RSS} = -2X^T y + 2X^T X \beta$$

Step 3: Set to zero and solve

$$\begin{aligned}-2X^T y + 2X^T X \beta &= 0 \\ X^T X \beta &= X^T y \\ \hat{\beta}_{\text{OLS}} &= (X^T X)^{-1} X^T y\end{aligned}$$

This is the **Ordinary Least Squares (OLS)** estimator. The equation $X^T X \beta = X^T y$ is called the **normal equations**.

Why “normal equations”? The name comes from the geometric interpretation: the residual vector $y - X\hat{\beta}$ is *normal* (perpendicular) to every column of X . We'll explore this geometry shortly.

NB!

[When OLS Fails] The OLS solution requires $X^T X$ to be invertible. This fails when:

- $n < p$ (more features than observations)—the system is underdetermined; infinitely many solutions exist
- Features are **perfectly collinear**—one feature is an exact linear combination of others (e.g., including both temperature in Celsius and Fahrenheit)
- Features are **near-collinear**—numerical instability leads to huge variance in estimates; small changes in data cause wild swings in $\hat{\beta}$

These issues motivate **regularisation** (Week 3): adding a penalty term that makes the problem well-posed even when $X^T X$ is singular or near-singular.

17.2 Geometric Interpretation: Orthogonal Projection

The OLS solution has a beautiful geometric interpretation that illuminates why it works and connects to fundamental ideas in linear algebra.

OLS as Orthogonal Projection

The fitted values $\hat{y} = X\hat{\beta}$ are the **orthogonal projection** of y onto the column space of X , denoted $\text{Col}(X)$.

The projection matrix (hat matrix):

$$H = X(X^\top X)^{-1}X^\top$$

Then $\hat{y} = Hy$ and the residuals are $e = y - \hat{y} = (I - H)y$.

Key properties of H :

- $H^2 = H$ (idempotent): projecting twice is the same as projecting once
- $H^\top = H$ (symmetric): orthogonal projection
- $\text{trace}(H) = p$: sum of “leverages” equals number of parameters

Why the name “hat matrix”? Because it puts the “hat” on y : $\hat{y} = Hy$. The matrix H transforms the observed values y into fitted values \hat{y} .

Geometric picture: Imagine y as a point in \mathbb{R}^n (one dimension per observation) and $\text{Col}(X)$ as a p -dimensional subspace (the set of all possible predictions $X\beta$ for any β). OLS finds the point in this subspace closest to y —which, by Euclidean geometry, is the orthogonal projection.

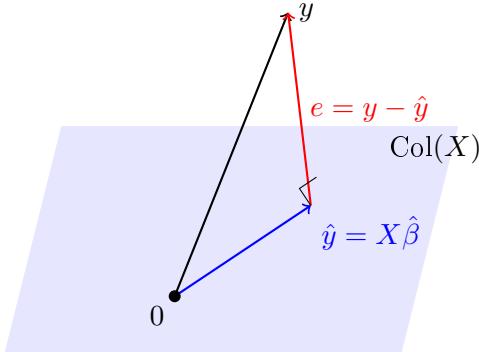


Figure 1: Geometric interpretation of OLS: \hat{y} is the orthogonal projection of y onto $\text{Col}(X)$, and the residual e is perpendicular to the column space.

Orthogonality of Residuals

The residuals are orthogonal to the column space of X :

$$X^\top e = X^\top(y - X\hat{\beta}) = X^\top y - X^\top X(X^\top X)^{-1}X^\top y = X^\top y - X^\top y = 0$$

This is the **normal equations** in disguise: $X^\top(y - X\hat{\beta}) = 0$.

Geometrically: the error vector is perpendicular to the space of possible predictions. Each column of X is orthogonal to e .

Why the Geometric View Matters

The projection interpretation:

1. **Explains uniqueness:** There is exactly one closest point in a subspace (assuming X has full column rank)
2. **Connects to Pythagoras:** $\|y\|^2 = \|\hat{y}\|^2 + \|e\|^2$ —total variation splits into explained and unexplained. This is because \hat{y} and e are orthogonal.
3. **Motivates R^2 :** The coefficient of determination $R^2 = \|\hat{y} - \bar{y}\|^2 / \|y - \bar{y}\|^2$ measures how much of y 's variation lies in $\text{Col}(X)$
4. **Generalises:** This view extends to ridge regression (regularised projection) and kernel methods

17.3 What OLS Gives Us

With $\hat{\beta} = (X^\top X)^{-1} X^\top y$, we can:

- **Predict:** $\hat{y} = X\hat{\beta}$ —fitted values for training data. For new data x_{new} , predict $\hat{y}_{\text{new}} = x_{\text{new}}^\top \hat{\beta}$.
- **Interpret:** $\hat{\beta}_j$ is the expected change in y for a unit change in x_j , *holding other features constant*. More precisely:

$$\frac{\partial \hat{y}}{\partial x_j} = \hat{\beta}_j$$

This “ceteris paribus” interpretation is crucial but often misunderstood—it’s about partial derivatives, not total effects.

- **Quantify uncertainty:** Via $\text{Var}(\hat{\beta})$ —how much would our estimates change with different data? This enables confidence intervals and hypothesis tests.

18 KL Divergence and Cross-Entropy

Section Summary

KL divergence $D_{\text{KL}}(p\|q)$ measures how “different” distribution q is from p . Intuition: the extra bits needed to encode samples from p using a code optimised for q . Crucially asymmetric. Minimising KL divergence from data to model is equivalent to MLE.

An alternative motivation for MLE comes from information theory: we want a model distribution q_θ that is “close” to the true data distribution p . But what does “close” mean for probability distributions? Ordinary Euclidean distance doesn’t make sense for distributions. KL divergence provides the answer.

18.1 Kullback-Leibler Divergence

Kullback-Leibler Divergence

The KL divergence from p to q measures how much information is lost when q is used to approximate p :

$$D_{\text{KL}}(p\|q) = \sum_y p(y) \log \frac{p(y)}{q(y)} = \mathbb{E}_{y \sim p} \left[\log \frac{p(y)}{q(y)} \right]$$

This can be decomposed as:

$$D_{\text{KL}}(p\|q) = \underbrace{-\sum_y p(y) \log p(y)}_{H(p)=\text{Entropy of } p} - \underbrace{\left(-\sum_y p(y) \log q(y) \right)}_{-H(p,q)=-\text{Cross-entropy}}$$

Rearranging: $D_{\text{KL}}(p\|q) = H(p, q) - H(p)$

For continuous distributions, replace sums with integrals:

$$D_{\text{KL}}(p\|q) = \int p(y) \log \frac{p(y)}{q(y)} dy$$

Properties:

- $D_{\text{KL}}(p\|q) \geq 0$ (Gibbs' inequality)
- $D_{\text{KL}}(p\|q) = 0$ if and only if $p = q$
- **Not symmetric:** $D_{\text{KL}}(p\|q) \neq D_{\text{KL}}(q\|p)$ in general

Unpacking the formula: Let's understand what each part means:

- $p(y)$ weights each outcome by its probability under the true distribution. We care more about getting common outcomes right than rare ones.
- $\log \frac{p(y)}{q(y)} = \log p(y) - \log q(y)$ compares the log-probability under p versus q . If $q(y) < p(y)$, this term is positive— q underestimates the probability of y .
- The expectation $\mathbb{E}_{y \sim p}[\cdot]$ averages over the true distribution p , not the model q .

18.2 Information-Theoretic Intuition

The KL divergence has a deep interpretation from coding theory that makes it more intuitive.

Background: To transmit data from a distribution p , the optimal code uses $-\log_2 p(y)$ bits for outcome y (Shannon's source coding theorem). Common outcomes get short codes; rare outcomes get long codes. The expected code length is the **entropy** $H(p) = -\sum_y p(y) \log_2 p(y)$.

The key insight: If we use a code optimised for distribution q but the data actually comes from p , we use $-\log q(y)$ bits for each outcome. The expected code length becomes the **cross-entropy** $H(p, q) = -\sum_y p(y) \log q(y)$.

KL Divergence as Extra Bits

$$D_{\text{KL}}(p\|q) = H(p, q) - H(p) = \text{Expected extra bits needed}$$

KL divergence measures the **coding inefficiency**—how many extra bits we waste by using a code optimised for q when the true distribution is p .

Example: If p is uniform over 8 outcomes, the optimal code uses $\log_2(8) = 3$ bits per outcome. If we mistakenly use a code optimised for a different distribution q , we'll use more than 3 bits on average. The excess is $D_{\text{KL}}(p\|q)$.

18.3 Why Asymmetry Matters

The asymmetry $D_{\text{KL}}(p\|q) \neq D_{\text{KL}}(q\|p)$ is not a defect but a feature with important consequences for machine learning.

Asymmetry of KL Divergence

Consider p as the “true” distribution and q as our model:

Forward KL: $D_{\text{KL}}(p\|q)$ — “mean-seeking” or “moment-matching”

- Penalises $q(y) \approx 0$ where $p(y) > 0$ (assigns low probability to likely events)
- The term $p(y) \log \frac{p(y)}{q(y)}$ explodes when $q(y) \rightarrow 0$ but $p(y) > 0$
- Forces q to cover all modes of p —can’t ignore any region where p has mass
- Used in MLE: we average over the true data distribution

Reverse KL: $D_{\text{KL}}(q\|p)$ — “mode-seeking”

- Penalises $q(y) > 0$ where $p(y) \approx 0$ (assigns probability to unlikely events)
- OK to have $q(y) = 0$ where $p(y) > 0$ (ignoring some modes is acceptable)
- Allows q to concentrate on one mode of p
- Used in variational inference: we average over the approximate distribution

Visualising the asymmetry: Let p be a bimodal distribution (two peaks). A unimodal q must choose:

- **Forward KL** ($D_{\text{KL}}(p\|q)$): q spreads to cover both modes, even if it assigns probability to the “valley” between them where p has little mass.
- **Reverse KL** ($D_{\text{KL}}(q\|p)$): q concentrates on one mode, ignoring the other entirely. This is acceptable because the penalty is proportional to $q(y)$, which is zero in the ignored region.

NB!

[Practical Implications of Asymmetry] The choice of KL direction profoundly affects learned models:

- **MLE** (forward KL) produces models that may be “overconfident” in regions between modes—assigning probability to outcomes that are actually unlikely under p
- **Variational inference** (reverse KL) may miss modes entirely, but is “honest” about uncertainty within the chosen mode

Understanding this asymmetry is essential for probabilistic modelling. Neither direction is “correct”—the choice depends on what failure mode is more acceptable for your application.

18.4 Cross-Entropy as a Loss Function

Cross-Entropy

The cross-entropy between true distribution p and model q :

$$H(p, q) = - \sum_y p(y) \log q(y)$$

For empirical data (where p is the empirical distribution putting mass $1/n$ on each training example):

$$H(p_{\text{data}}, q_{\theta}) = -\frac{1}{n} \sum_{i=1}^n \log q_{\theta}(y_i)$$

This is exactly the NLL (up to a constant factor of n).

MLE Minimises KL Divergence

Since the entropy of the true distribution $H(p)$ is constant (doesn’t depend on model parameters), minimising KL divergence is equivalent to minimising cross-entropy:

$$\arg \min_{\theta} D_{\text{KL}}(p \| q_{\theta}) = \arg \min_{\theta} H(p, q_{\theta}) = \arg \min_{\theta} \text{NLL}(\theta)$$

Key insight: MLE finds the model closest to the data in the KL sense. This gives MLE an information-theoretic justification beyond just “maximising probability.”

19 Variance of the OLS Estimator

Section Summary

The OLS estimator $\hat{\beta}$ is unbiased with variance in “sandwich form”: $\text{Var}(\hat{\beta}) = (X^T X)^{-1} X^T \mathbb{E}[\epsilon \epsilon^T] X (X^T X)^{-1}$. Under homoskedasticity this simplifies to $\sigma^2 (X^T X)^{-1}$. Robust standard errors handle heteroskedasticity by estimating the error covariance from residuals.

To do inference (hypothesis tests, confidence intervals), we need the sampling distribution of $\hat{\beta}$.

The key question: if we collected new data and recomputed $\hat{\beta}$, how much would it vary?

19.1 Deriving the Variance

Starting from $\hat{\beta} = (X^\top X)^{-1} X^\top y$ and substituting $y = X\beta + \epsilon$:

$$\begin{aligned}\hat{\beta} &= (X^\top X)^{-1} X^\top (X\beta + \epsilon) \\ &= (X^\top X)^{-1} X^\top X\beta + (X^\top X)^{-1} X^\top \epsilon \\ &= \beta + (X^\top X)^{-1} X^\top \epsilon\end{aligned}$$

This decomposition is illuminating: $\hat{\beta}$ equals the true β plus a random perturbation $(X^\top X)^{-1} X^\top \epsilon$ that depends on the errors. The perturbation has mean zero (since $\mathbb{E}[\epsilon] = 0$), so $\hat{\beta}$ is unbiased.

Unbiasedness:

$$\mathbb{E}[\hat{\beta}] = \beta + (X^\top X)^{-1} X^\top \mathbb{E}[\epsilon] = \beta + 0 = \beta$$

So OLS is an **unbiased estimator**—on average, across many datasets, it gives the right answer.

Sandwich Form of Variance

$$\begin{aligned}\text{Var}(\hat{\beta}) &= \mathbb{E}[(\hat{\beta} - \beta)(\hat{\beta} - \beta)^\top] \\ &= \mathbb{E}[(X^\top X)^{-1} X^\top \epsilon \epsilon^\top X (X^\top X)^{-1}] \\ &= (X^\top X)^{-1} X^\top \mathbb{E}[\epsilon \epsilon^\top] X (X^\top X)^{-1}\end{aligned}$$

This is the **sandwich estimator**:

$$\text{Var}(\hat{\beta}) = \underbrace{(X^\top X)^{-1}}_{\text{bread}} \underbrace{X^\top \mathbb{E}[\epsilon \epsilon^\top]}_{\text{meat}} \underbrace{X (X^\top X)^{-1}}_{\text{bread}}$$

Why “sandwich”? The covariance matrix of errors $\mathbb{E}[\epsilon \epsilon^\top]$ (the “meat”) is sandwiched between two copies of $(X^\top X)^{-1} X^\top$ and its transpose (the “bread”). This structure appears throughout statistics whenever we transform random variables by a linear operation.

The “meat” $\mathbb{E}[\epsilon \epsilon^\top]$ is the covariance matrix of the errors. Its structure depends on our assumptions about the error distribution.

19.2 Homoskedastic Errors

Variance Under Homoskedasticity

If errors have constant variance and are uncorrelated: $\mathbb{E}[\epsilon\epsilon^\top] = \sigma^2 I$

Then the sandwich simplifies dramatically:

$$\begin{aligned}\text{Var}(\hat{\beta}) &= (X^\top X)^{-1} X^\top \sigma^2 I \cdot X(X^\top X)^{-1} \\ &= \sigma^2 (X^\top X)^{-1} X^\top X (X^\top X)^{-1} \\ &= \sigma^2 (X^\top X)^{-1}\end{aligned}$$

The variance σ^2 is estimated by:

$$\hat{\sigma}^2 = \frac{1}{n-p} \sum_{i=1}^n (y_i - x_i^\top \hat{\beta})^2 = \frac{\text{RSS}}{n-p}$$

(We divide by $n-p$ rather than n for unbiasedness—we “used up” p degrees of freedom estimating β .)

The **standard error** of $\hat{\beta}_j$ is: $\text{SE}(\hat{\beta}_j) = \hat{\sigma} \sqrt{[(X^\top X)^{-1}]_{jj}}$

Unpacking the homoskedasticity assumption: The condition $\mathbb{E}[\epsilon\epsilon^\top] = \sigma^2 I$ means:

- **Constant variance:** $\text{Var}(\epsilon_i) = \sigma^2$ for all i . The spread of errors is the same regardless of x_i .
- **Uncorrelated errors:** $\text{Cov}(\epsilon_i, \epsilon_j) = 0$ for $i \neq j$. Knowing one error tells us nothing about another.

19.3 Heteroskedastic Errors

When error variance varies across observations, the homoskedastic formula understates uncertainty for some coefficients and overstates it for others. We need **robust standard errors**.

Heteroskedasticity-Consistent (HC) Standard Errors

If $\text{Var}(\epsilon_i) = \sigma_i^2$ (varies with i), we can't simplify the sandwich. Instead, we estimate the meat directly:

$$\mathbb{E}[\epsilon\epsilon^\top] \approx \text{diag}(\hat{e}_1^2, \dots, \hat{e}_n^2)$$

where $\hat{e}_i = y_i - x_i^\top \hat{\beta}$ are the residuals.

The **HC0 estimator** (White's robust standard errors):

$$\widehat{\text{Var}}(\hat{\beta}) = (X^\top X)^{-1} X^\top \text{diag}(\hat{e}^2) X (X^\top X)^{-1}$$

To compute robust standard errors:

1. Calculate residuals: $\hat{e}_i = y_i - x_i^\top \hat{\beta}$
2. Form the diagonal matrix $\Omega = \text{diag}(\hat{e}_1^2, \dots, \hat{e}_n^2)$
3. Compute the sandwich: $(X^\top X)^{-1} X^\top \Omega X (X^\top X)^{-1}$
4. Take square roots of diagonal elements to get standard errors

When to Use Robust Standard Errors

- **Always safe:** Robust SEs are valid under both homo- and heteroskedasticity
- **Efficiency:** If errors truly are homoskedastic, classical SEs are more efficient (lower variance)
- **Practice:** Many applied fields default to robust SEs as insurance
- **Individual variances:** Diagonal elements give variance of each $\hat{\beta}_j$; off-diagonals give covariances between coefficient estimates

Significance of the Variance of $\hat{\beta}$

The variance-covariance matrix of $\hat{\beta}$ enables:

1. **Statistical Significance:** Test whether coefficients differ from zero (or other values)
2. **Confidence Intervals:** Quantify uncertainty in coefficient estimates
3. **Precision Assessment:** Smaller variance = more precise estimates
4. **Model Diagnostics:** High variance may indicate collinearity or insufficient data
5. **Covariance Information:** Off-diagonal elements show how uncertainty in one coefficient relates to uncertainty in another

20 Bayesian Inference and MAP Estimation

Section Summary

Bayesian inference treats parameters as random variables with a prior $p(\theta)$. Bayes' rule combines prior and likelihood to give the posterior $p(\theta|\mathcal{D})$. MAP estimation finds the posterior mode; with Gaussian priors this yields Ridge regression, with Laplace priors it yields Lasso.

MLE treats parameters as fixed unknowns. **Bayesian inference** treats parameters as random variables with distributions reflecting uncertainty. This is a fundamentally different philosophical stance with practical implications.

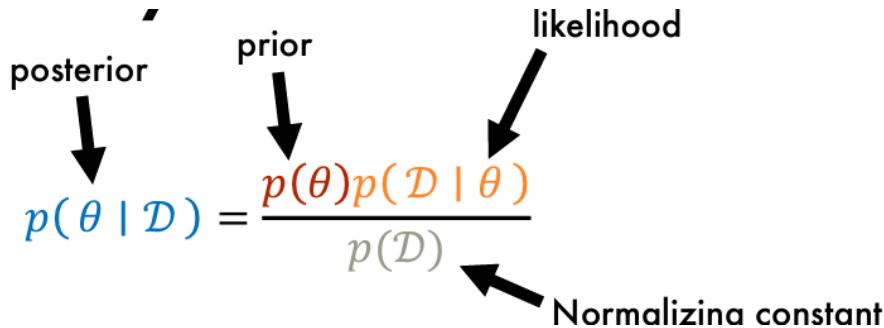


Figure 2: Bayes' Rule: combining prior beliefs with observed data to form posterior beliefs. The posterior balances what we believed before (prior) with what the data tells us (likelihood).

Bayes' Rule for Parameter Inference

$$p(\theta|\mathcal{D}) = \frac{p(\mathcal{D}|\theta) \cdot p(\theta)}{p(\mathcal{D})}$$

- $p(\theta|\mathcal{D})$: **Posterior**—belief about θ after seeing data
- $p(\mathcal{D}|\theta)$: **Likelihood**—probability of data given parameters (same as MLE)
- $p(\theta)$: **Prior**—belief about θ before seeing data
- $p(\mathcal{D})$: **Marginal likelihood** (evidence)—normalising constant ensuring the posterior integrates to 1

Unpacking Bayes' rule:

- The **prior** $p(\theta)$ encodes what we believe about the parameters before seeing any data. This could be informed by previous studies, domain knowledge, or simply a “weakly informative” default.
- The **likelihood** $p(\mathcal{D}|\theta)$ is exactly what MLE uses—the probability of the observed data under each possible parameter value.
- The **posterior** $p(\theta|\mathcal{D})$ combines both sources of information. It tells us what we should believe about θ after seeing the data.
- The **marginal likelihood** $p(\mathcal{D}) = \int p(\mathcal{D}|\theta)p(\theta)d\theta$ is a normalising constant. For point estimation (MAP), we can ignore it.

Frequentist vs Bayesian Perspectives

	Frequentist (MLE)	Bayesian
Parameters	Fixed, unknown constants	Random variables
Data	Random (from repeated sampling)	Fixed (what we observed)
Probability	Long-run frequency	Degree of belief
Primary quantity	$p(\mathcal{D} \theta)$ (likelihood)	$p(\theta \mathcal{D})$ (posterior)

Objective probability (frequentist): Based on facts about properties of the world. “If we flip this coin infinitely many times, what fraction will be heads?”

Subjective probability (Bayesian): Based on our beliefs about the world. “Given what I know, how confident am I that this coin is fair?”

20.1 Maximum A Posteriori (MAP) Estimation

Full Bayesian inference requires computing the entire posterior distribution, which can be computationally demanding (often requiring Monte Carlo methods). MAP estimation finds just the *mode* of the posterior—the single most probable parameter value.

MAP Estimation

$$\hat{\theta}_{\text{MAP}} = \arg \max_{\theta} p(\theta|\mathcal{D}) = \arg \max_{\theta} \frac{p(\mathcal{D}|\theta) \cdot p(\theta)}{p(\mathcal{D})}$$

Since $p(\mathcal{D})$ doesn't depend on θ :

$$\hat{\theta}_{\text{MAP}} = \arg \max_{\theta} [p(\mathcal{D}|\theta) \cdot p(\theta)]$$

Taking logs (for computational convenience):

$$\hat{\theta}_{\text{MAP}} = \arg \max_{\theta} [\log p(\mathcal{D}|\theta) + \log p(\theta)]$$

This is MLE plus a **regularisation term** from the prior.

The key insight: MAP = MLE + Prior. The log-prior acts as a penalty term that discourages certain parameter values. Different priors give different penalties:

Connection Between MAP and Regularisation

The choice of prior determines the type of regularisation:

- **Gaussian prior:** $p(\theta) \propto \exp(-\lambda\|\theta\|_2^2) \Rightarrow$ **Ridge regression** (L2 penalty)

Taking logs: $\log p(\theta) = -\lambda\|\theta\|_2^2 + \text{const}$

- **Laplace prior:** $p(\theta) \propto \exp(-\lambda\|\theta\|_1) \Rightarrow$ **Lasso** (L1 penalty)

Taking logs: $\log p(\theta) = -\lambda\|\theta\|_1 + \text{const}$

- **Uniform/flat prior** (improper): $p(\theta) \propto 1 \Rightarrow \text{MAP} = \text{MLE}$

The log-prior is constant, so it doesn't affect the optimisation.

Regularisation isn't just a computational trick—it has a probabilistic interpretation as encoding prior beliefs about parameter values.

20.2 Interpreting Uncertainty: Credible vs Confidence Intervals

With $\text{Var}(\hat{\beta})$, we can construct intervals. The interpretation differs fundamentally between paradigms:

Bayesian (Credible Intervals):

- Assume β is random, data is fixed
- “There is a 95% probability that $\beta \in [l, u]$ ”
- Direct probability statement about the parameter
- Based on the posterior distribution $p(\beta|\mathcal{D})$

Frequentist (Confidence Intervals):

- Assume β is fixed, data is random
- “If we repeated this experiment many times, 95% of the constructed intervals would contain the true β ”
- Statement about the *procedure*, not the parameter
- We don't assume a distribution for β ; instead, we think about a process for constructing intervals

Confidence Interval Construction

$$[\hat{\beta} - z_{\alpha/2} \cdot \text{SE}(\hat{\beta}), \quad \hat{\beta} + z_{\alpha/2} \cdot \text{SE}(\hat{\beta})]$$

For 95% confidence: $z_{0.025} \approx 1.96$

With finite samples and unknown σ^2 , use t -distribution critical values instead:

$$[\hat{\beta} - t_{n-p,\alpha/2} \cdot \text{SE}(\hat{\beta}), \quad \hat{\beta} + t_{n-p,\alpha/2} \cdot \text{SE}(\hat{\beta})]$$

NB!

[Common Misinterpretation] A 95% confidence interval does **not** mean “there is a 95% probability that β is in this interval.” That’s a Bayesian statement!

The frequentist statement is: “The procedure that generated this interval has a 95% success rate.” The specific interval either contains β or it doesn’t—we just don’t know which.

This distinction matters in practice: if you want to make probability statements about parameters, you need the Bayesian framework.

21 Empirical Risk Minimisation

Section Summary

ERM generalises MLE: minimise average loss $\frac{1}{n} \sum_i \ell(y_i, f(x_i; \theta))$ over training data. Different losses suit different tasks. For classification, 0-1 loss is natural but non-convex; surrogate losses (log loss, hinge) are convex upper bounds that enable optimisation.

MLE is one instance of a broader framework: **Empirical Risk Minimisation (ERM)**. The idea is simple: define a loss function that measures prediction error, then minimise the average loss on training data.

Empirical Risk Minimisation

Given a loss function $\ell(y, \hat{y})$ measuring prediction error:

$$\hat{\theta} = \arg \min_{\theta} \frac{1}{n} \sum_{i=1}^n \ell(y_i, f(x_i; \theta))$$

This minimises the **empirical risk**—the average loss on training data.

Common loss functions include:

- **NLL (for MLE)**: $\ell(y, \theta; x) = -\log p(y|x, \theta)$
- **Squared error**: $\ell(y, \hat{y}) = (y - \hat{y})^2$
- **0-1 loss**: $\ell(y, \hat{y}) = \mathbf{1}[y \neq \hat{y}]$

Why “empirical” risk? The true risk is the expected loss over the true data distribution: $R(\theta) = \mathbb{E}_{(x,y) \sim p}[\ell(y, f(x; \theta))]$. We can’t compute this because we don’t know p . The empirical risk approximates it using the training data as a sample from p .

21.1 Common Loss Functions

Loss	Formula	Use Case
Squared (L2)	$(y - \hat{y})^2$	Regression
Absolute (L1)	$ y - \hat{y} $	Robust regression
NLL (Gaussian)	$-\log p(y \hat{y}, \sigma)$	Probabilistic regression
0-1 Loss	$\mathbf{1}[y \neq \hat{y}]$	Classification (ideal)
Log Loss (Cross-entropy)	$-y \log \hat{p} - (1 - y) \log(1 - \hat{p})$	Probabilistic classification
Hinge Loss	$\max(0, 1 - y \cdot \hat{y})$	SVM classification

Why different losses? Each loss encodes different priorities:

- **L2 loss:** Penalises large errors heavily (squared). Sensitive to outliers.
- **L1 loss:** Penalises errors linearly. More robust to outliers.
- **0-1 loss:** Only cares about correct/incorrect, not confidence. The natural classification metric.
- **Log loss:** Rewards well-calibrated probability estimates. Penalises confident wrong predictions severely.
- **Hinge loss:** Focuses on margin—how far points are from the decision boundary.

21.2 The Problem with 0-1 Loss

NB!

[0-1 Loss is Intractable] The **0-1 loss** (misclassification rate) is the natural classification loss—it directly measures what we care about. But it is:

- **Non-convex:** Has many local minima; gradient descent may get stuck
- **Non-differentiable:** The function jumps from 0 to 1 at the decision boundary
- **Flat almost everywhere:** Gradient is zero except at decision boundary, providing no signal for optimisation

We cannot optimise it with gradient methods. Instead, we use **surrogate losses**.

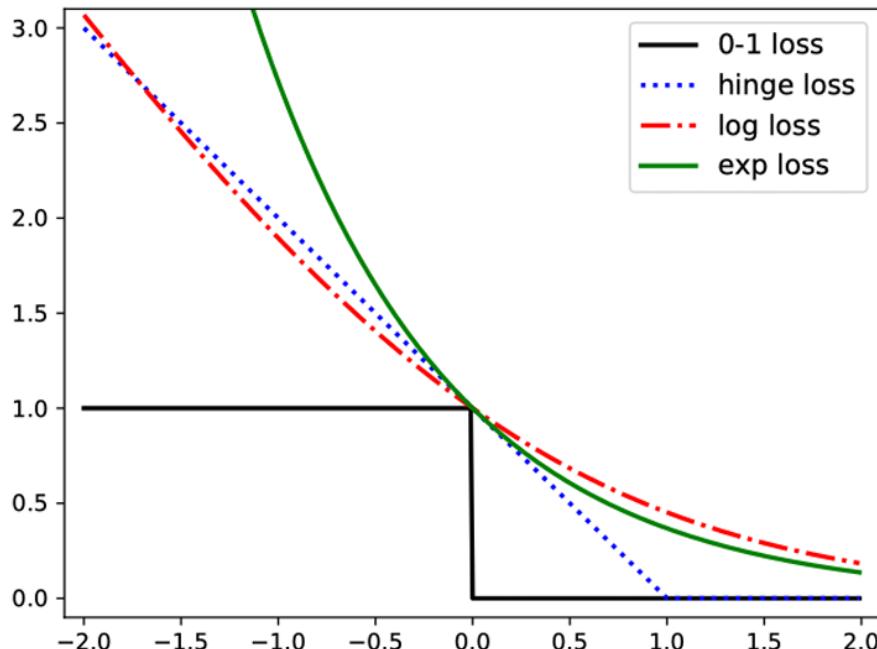


Figure 3: Surrogate loss functions compared to 0-1 loss. All surrogate losses upper bound the 0-1 loss while being differentiable and (mostly) convex.

Surrogate losses replace the 0-1 loss with something we can actually optimise. A good surrogate should:

1. **Upper bound** the 0-1 loss: $\ell_{\text{surrogate}}(y, \hat{y}) \geq \mathbf{1}[y \neq \hat{y}]$. This ensures that minimising the surrogate also reduces misclassification.
2. Be **convex**: Guarantees finding global minimum; no local minima traps.
3. Be **differentiable**: Enables gradient-based optimisation.
4. Be “tight”: Close to 0-1 loss, so the approximation is good.

21.3 Classification Errors

		Predicted	
		Positive	Negative
Actual	Positive	TP (True Positive)	FN (False Negative)
	Negative	FP (False Positive)	TN (True Negative)

- **Type I Error** (False Positive): Predict positive when actually negative. “False alarm.”
- **Type II Error** (False Negative): Predict negative when actually positive. “Missed detection.”

		True	
		0	1
Predicted	0	✓	Type II
	1	Type I	✓

Figure 4: Confusion matrix structure showing the four possible outcomes of binary classification.

Different applications weight these errors differently:

- **Medical screening**: False negatives are dangerous (missing disease), so we tolerate more false positives. High **sensitivity** (recall) is prioritised.
- **Spam filtering**: False positives are annoying (good email marked as spam), so we tolerate more false negatives. High **precision** is prioritised.
- **Criminal justice**: “Beyond reasonable doubt” means we prefer false negatives (letting guilty go free) to false positives (convicting innocent).

22 Convexity and Optimisation

Section Summary

Convex loss functions have a unique global minimum and no local minima traps. Gradient descent iteratively moves downhill in the loss landscape. For convex problems with appropriate step sizes, gradient descent is guaranteed to converge. Non-convex problems (neural networks) require more sophisticated approaches.

When closed-form solutions don't exist (logistic regression, neural networks), we rely on iterative optimisation. The structure of the loss function determines how hard this is.

22.1 Convex Functions

Convex Function

A function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ is **convex** if for all $x, y \in \mathbb{R}^d$ and $\lambda \in [0, 1]$:

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$$

Geometrically: the line segment connecting any two points on the graph lies above (or on) the graph.

Equivalent characterisations (for twice-differentiable f):

- First-order: $f(y) \geq f(x) + \nabla f(x)^\top (y - x)$ for all x, y
(The tangent plane at any point lies below the function.)
- Second-order: $\nabla^2 f(x) \succeq 0$ for all x
(The Hessian is positive semi-definite everywhere—the function curves upward.)

Unpacking convexity: The defining inequality says: if you take a weighted average of two inputs ($\lambda x + (1 - \lambda)y$), the function value there is at most the weighted average of the function values ($\lambda f(x) + (1 - \lambda)f(y)$). Visually, this means the function “curves upward” like a bowl.

Why Convexity Matters

For convex functions:

1. **Local = Global:** Any local minimum is a global minimum. No risk of getting stuck in suboptimal valleys.
2. **Unique minimiser:** If the function is strictly convex (strict inequality), there is exactly one minimiser.
3. **Guarantees:** We can prove convergence of optimisation algorithms.
4. **Efficiency:** Polynomial-time algorithms exist for finding the minimum.

Examples of convex losses:

- **Squared loss:** $(y - \hat{y})^2$ —second derivative is $2 > 0$
- **Log loss:** $-y \log \hat{p} - (1 - y) \log(1 - \hat{p})$ —can show Hessian is PSD

- **Hinge loss:** $\max(0, 1 - y \cdot \hat{y})$ —convex but non-differentiable at the “hinge”

NB!

[Non-Convexity in Deep Learning] The 0-1 loss is **not convex**. Neither are neural network loss surfaces (due to nonlinear activations and the composition of many layers).

Deep learning succeeds despite non-convexity, but lacks the theoretical guarantees of convex optimisation. Current understanding suggests:

- Many local minima exist, but they’re often nearly as good as the global minimum
- Saddle points (not local minima) are the main obstacle
- Overparameterisation may help optimisation by creating more paths to good solutions

22.2 Gradient Descent

When no closed-form solution exists, we use iterative methods. Gradient descent is the workhorse of machine learning optimisation.

Gradient Descent

To minimise $\mathcal{L}(\theta)$, iterate:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_{\theta} \mathcal{L}(\theta^{(t)})$$

where $\eta > 0$ is the **learning rate** (step size).

Intuition: The negative gradient $-\nabla \mathcal{L}$ points in the direction of steepest descent. We take a step of size η in that direction.

Geometric picture: Imagine standing on a hilly landscape where height represents loss. The gradient tells you which way is uphill (steepest ascent). Gradient descent walks downhill, step by step, following the direction of steepest descent.

Unpacking the update rule:

- $\nabla_{\theta} \mathcal{L}(\theta^{(t)})$ is the gradient—a vector pointing in the direction of steepest increase.
- The negative sign reverses this to point downhill.
- η scales the step size. Too small: slow progress. Too large: overshoot and oscillate.

Convergence for Convex Functions

For a convex function \mathcal{L} with **L -Lipschitz gradients** (i.e., $\|\nabla \mathcal{L}(x) - \nabla \mathcal{L}(y)\| \leq L \|x - y\|$):

With step size $\eta = 1/L$, gradient descent achieves:

$$\mathcal{L}(\theta^{(T)}) - \mathcal{L}(\theta^*) \leq \frac{L \|\theta^{(0)} - \theta^*\|^2}{2T}$$

This is $O(1/T)$ convergence: to get within ϵ of the optimum, we need $O(1/\epsilon)$ iterations.

For **strongly convex** functions (Hessian bounded below: $\nabla^2 \mathcal{L} \succeq \mu I$ for $\mu > 0$), convergence is exponentially faster: $O(\log(1/\epsilon))$ iterations.

What is Lipschitz continuity of gradients? It means the gradient doesn't change too rapidly. If the gradient changes slowly (small L), we can take larger steps safely. If it changes rapidly (large L), we need smaller steps to avoid overshooting.

22.3 Choosing the Learning Rate

Learning Rate Selection

- **Too small:** Slow convergence; may take forever to reach minimum
- **Too large:** Overshoots; may diverge (oscillate around minimum or explode)
- **Just right:** Fast, stable convergence

Rules of thumb:

- Start with $\eta = 0.01$ or 0.001
- Use learning rate schedules (decay over time): start large, decrease as you approach the minimum
- Adaptive methods (Adam, RMSprop) adjust per-parameter automatically

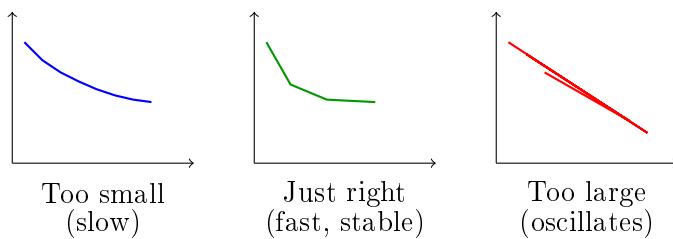


Figure 5: Effect of learning rate on gradient descent convergence. Loss versus iteration for different learning rates.

22.4 Stochastic Gradient Descent (SGD)

For large datasets, computing the full gradient is expensive—it requires a pass through all n samples.

Stochastic Gradient Descent

Instead of using all n samples:

$$\nabla \mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^n \nabla \ell_i(\theta)$$

Use a random **minibatch** $\mathcal{B} \subset \{1, \dots, n\}$:

$$\nabla \mathcal{L}(\theta) \approx \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla \ell_i(\theta)$$

This is an unbiased estimate of the true gradient: $\mathbb{E}[\text{minibatch gradient}] = \text{full gradient}$.

The variance decreases with batch size: $\text{Var} \propto 1/|\mathcal{B}|$.

SGD Tradeoffs

- **Computation:** Each step is $O(|\mathcal{B}|)$ instead of $O(n)$ —much faster per iteration
- **Noise:** Stochastic gradients are noisy estimates; updates are “wiggly”
- **Regularisation:** Noise can help escape sharp minima (implicit regularisation)—may actually improve generalisation
- **Convergence:** More iterations needed, but much faster wall-clock time for large n

Typical batch sizes: 32, 64, 128, 256. Larger = more accurate gradients but slower per-epoch progress.

23 Logistic Regression

Section Summary

Logistic regression models $p(y = 1|x) = \sigma(x^\top \beta)$ where σ is the sigmoid. Trained by minimising binary cross-entropy (NLL for Bernoulli). No closed-form solution—requires iterative optimisation. Decision boundary is linear: the hyperplane $x^\top \beta = 0$.

For binary classification, we need a model that outputs probabilities in $[0, 1]$. Linear regression won’t work—it can produce any real number. We need a function that “squashes” the linear predictor into the valid probability range.

Logistic Regression Model

$$p(y = 1|x, \beta) = \sigma(x^\top \beta) = \frac{1}{1 + e^{-x^\top \beta}}$$

where $\sigma(\cdot)$ is the **sigmoid** (logistic) function.

Equivalently, the observation follows a Bernoulli distribution:

$$y_i \sim \text{Bernoulli}(\sigma(x_i^\top \beta))$$

The **log-odds** (logit) is linear in the features:

$$\log \frac{p(y = 1|x)}{p(y = 0|x)} = \log \frac{p(y = 1|x)}{1 - p(y = 1|x)} = x^\top \beta$$

Unpacking the model:

- $x^\top \beta$ is the linear predictor, same as in linear regression. It can be any real number.
- $\sigma(z) = \frac{1}{1+e^{-z}}$ squashes this to $(0, 1)$. Large positive z gives probability near 1; large negative z gives probability near 0.
- The log-odds being linear means: each unit increase in x_j adds β_j to the log-odds of $y = 1$.

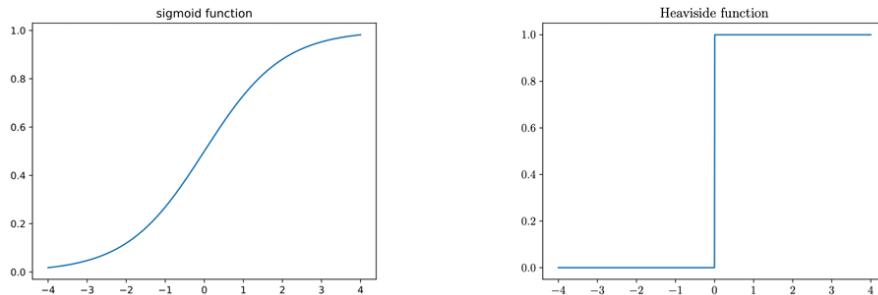


Figure 6: The sigmoid (logistic) function $\sigma(z) = 1/(1 + e^{-z})$ maps any real number to $(0, 1)$. At $z = 0$, $\sigma(0) = 0.5$. The function saturates at 0 and 1 for large $|z|$.

The sigmoid function has useful properties:

- **Range:** Maps $\mathbb{R} \rightarrow (0, 1)$ —valid probabilities
- **Symmetric:** $\sigma(-z) = 1 - \sigma(z)$
- **Nice derivative:** $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ —easy to compute from the function value
- **Monotonic:** Larger $x^\top \beta$ means higher probability of $y = 1$

23.1 Training: Binary Cross-Entropy

Binary Cross-Entropy Loss

The NLL for logistic regression (also called “log loss” or “binary cross-entropy”):

Starting from the Bernoulli likelihood for one observation:

$$p(y_i|x_i, \beta) = \mu_i^{y_i} (1 - \mu_i)^{1-y_i}$$

where $\mu_i = \sigma(x_i^\top \beta)$ and $y_i \in \{0, 1\}$.

Taking the negative log and averaging:

$$\begin{aligned} \text{NLL}(\beta) &= -\frac{1}{n} \sum_{i=1}^n \log [\mu_i^{y_i} (1 - \mu_i)^{1-y_i}] \\ &= -\frac{1}{n} \sum_{i=1}^n [y_i \log \mu_i + (1 - y_i) \log(1 - \mu_i)] \end{aligned}$$

Gradient:

$$\nabla_\beta \text{NLL} = \frac{1}{n} \sum_{i=1}^n (\mu_i - y_i) x_i = \frac{1}{n} X^\top (\mu - y)$$

This has the same form as linear regression’s gradient! But μ depends nonlinearly on β through the sigmoid.

Unpacking the loss function: For each observation:

- If $y_i = 1$: loss is $-\log \mu_i$. We want μ_i (predicted probability of 1) to be high.
- If $y_i = 0$: loss is $-\log(1 - \mu_i)$. We want μ_i to be low (so $1 - \mu_i$ is high).

The loss penalises confident wrong predictions severely: if $y_i = 1$ but $\mu_i \approx 0$, then $-\log \mu_i \rightarrow \infty$.

NB!

[No Closed-Form Solution] Unlike linear regression, logistic regression has **no closed-form solution**. The dependence of μ_i on β through the sigmoid makes the normal equations nonlinear. Setting the gradient to zero gives:

$$\sum_{i=1}^n (\sigma(x_i^\top \beta) - y_i) x_i = 0$$

This cannot be solved algebraically for β . We must use iterative optimisation:

- **Gradient descent**: Simple but may be slow
- **Newton’s method** (IRLS—Iteratively Reweighted Least Squares): Faster convergence using second derivatives
- **Quasi-Newton methods** (L-BFGS): Approximate Newton without computing full Hessian

23.2 Decision Boundaries

The classifier predicts $\hat{y} = 1$ when $p(y = 1|x) > 0.5$. Since $\sigma(0) = 0.5$, this happens when $x^\top \beta > 0$.

Linear Decision Boundary

The decision boundary $\{x : x^\top \beta = 0\}$ is a **hyperplane** in feature space.

- In 2D (two features): the boundary is a line
- In 3D: a plane
- In general: a $(p - 1)$ -dimensional hyperplane in p -dimensional space

Logistic regression is a **linear classifier**—it can only separate classes with a linear boundary. For nonlinear boundaries, we need:

- **Feature engineering:** Add polynomial features, interactions (e.g., x_1^2 , $x_1 x_2$)
- **Kernel methods:** Implicitly map to high-dimensional feature space
- **Neural networks:** Learn nonlinear transformations of features

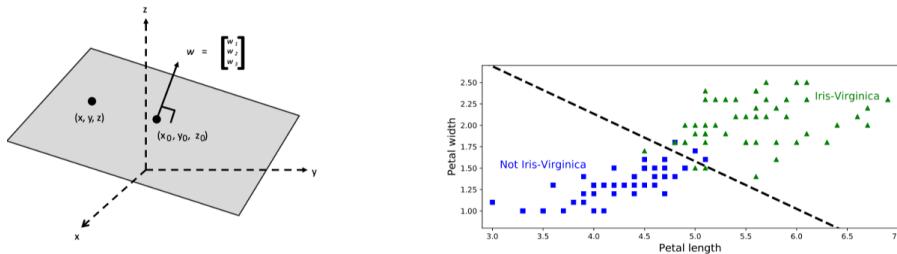


Figure 7: Linear decision boundary separating two classes. The boundary is the hyperplane where $x^\top \beta = 0$. Points on one side are classified as positive, points on the other as negative.

24 The Bias-Variance Tradeoff

Section Summary

Expected prediction error decomposes as $\text{MSE} = \text{Bias}^2 + \text{Variance} + \text{Irreducible noise}$. Simple models have high bias (systematic error), complex models have high variance (sensitivity to training data). Optimal model complexity balances both. This motivates regularisation.

A fundamental tension in statistical learning: simple models underfit (high bias), complex models overfit (high variance). Understanding this tradeoff is crucial for building models that generalise well.

Bias-Variance Decomposition for Estimators

For an estimator $\hat{\theta}$ of true parameter θ :

$$\text{Bias: } \text{Bias}(\hat{\theta}) = \mathbb{E}[\hat{\theta}] - \theta$$

How far is the average estimate from the truth? Measures **systematic error**.

$$\text{Variance: } \text{Var}(\hat{\theta}) = \mathbb{E}[(\hat{\theta} - \mathbb{E}[\hat{\theta}])^2]$$

How much does the estimate vary across different datasets? Measures **instability**.

MSE Decomposition:

$$\text{MSE}(\hat{\theta}) = \mathbb{E}[(\hat{\theta} - \theta)^2] = \text{Var}(\hat{\theta}) + \text{Bias}(\hat{\theta})^2$$

24.1 Full Derivation of the Decomposition

The MSE decomposition is fundamental. Let us derive it carefully using the “add and subtract the mean” trick.

Derivation of Bias-Variance Decomposition

Let $\bar{\theta} = \mathbb{E}[\hat{\theta}]$ denote the expected value of our estimator. We use the “add and subtract” trick:

$$\begin{aligned} \text{MSE}(\hat{\theta}) &= \mathbb{E}[(\hat{\theta} - \theta)^2] \\ &= \mathbb{E}[(\hat{\theta} - \bar{\theta} + \bar{\theta} - \theta)^2] \quad (\text{add and subtract } \bar{\theta}) \\ &= \mathbb{E}[(\hat{\theta} - \bar{\theta})^2 + 2(\hat{\theta} - \bar{\theta})(\bar{\theta} - \theta) + (\bar{\theta} - \theta)^2] \end{aligned}$$

Now we take expectations term by term:

- **First term:** $\mathbb{E}[(\hat{\theta} - \bar{\theta})^2] = \text{Var}(\hat{\theta})$ by definition of variance
- **Second term:** $\mathbb{E}[2(\hat{\theta} - \bar{\theta})(\bar{\theta} - \theta)] = 2(\bar{\theta} - \theta)\mathbb{E}[\hat{\theta} - \bar{\theta}] = 2(\bar{\theta} - \theta) \cdot 0 = 0$
This is zero because $\mathbb{E}[\hat{\theta}] = \bar{\theta}$ by definition, so $\mathbb{E}[\hat{\theta} - \bar{\theta}] = 0$.
- **Third term:** $\mathbb{E}[(\bar{\theta} - \theta)^2] = (\bar{\theta} - \theta)^2 = \text{Bias}(\hat{\theta})^2$
This is a constant (not random), so expectation doesn’t change it.

Therefore:

$$\boxed{\text{MSE}(\hat{\theta}) = \text{Var}(\hat{\theta}) + \text{Bias}(\hat{\theta})^2}$$

24.2 Prediction Error Decomposition

For prediction tasks, we decompose the expected prediction error at a test point x_0 :

Bias-Variance-Noise Decomposition for Prediction

Assume the true model is $y = f(x) + \epsilon$ where $\epsilon \sim (0, \sigma^2)$ is irreducible noise. Let $\hat{f}(x)$ be our learned predictor (a random variable depending on training data).

The expected prediction error at x_0 :

$$\begin{aligned}\mathbb{E}[(y_0 - \hat{f}(x_0))^2] &= \mathbb{E}[(f(x_0) + \epsilon - \hat{f}(x_0))^2] \\ &= \underbrace{\sigma^2}_{\text{irreducible}} + \underbrace{(f(x_0) - \mathbb{E}[\hat{f}(x_0)])^2}_{\text{bias}^2} + \underbrace{\mathbb{E}[(\hat{f}(x_0) - \mathbb{E}[\hat{f}(x_0)])^2]}_{\text{variance}}\end{aligned}$$

The expectation is over both the noise ϵ and the randomness in \hat{f} from different training sets.

The three components:

1. **Irreducible error (σ^2)**: Noise inherent in the problem; cannot be reduced by any model, no matter how sophisticated. This is the “floor” on our prediction error.
2. **Bias²**: How far the average prediction is from the truth; measures systematic error. High bias means the model is “too simple” to capture the true relationship.
3. **Variance**: How much predictions vary across different training sets; measures instability. High variance means the model is “too complex” and fits noise in the training data.

24.3 Visual Intuition: The Bullseye

The bias-variance tradeoff has an intuitive interpretation as a target-shooting analogy:

	Low Variance	High Variance
Low Bias	Tight cluster on bullseye <i>(ideal)</i>	Scattered around bullseye <i>(overfitting)</i>
High Bias	Tight cluster off-centre <i>(underfitting)</i>	Scattered off-centre <i>(worst case)</i>

- **Bias** = how far the centre of mass of shots is from the bullseye
- **Variance** = how spread out the shots are around their centre of mass
- **MSE** = average squared distance from shots to bullseye = bias² + variance

The Tradeoff

- **Low bias, high variance**: Complex models (many parameters) fit training data well but vary wildly between samples—they **overfit**
- **High bias, low variance**: Simple models (few parameters) are stable but systematically wrong—they **underfit**
- **Optimal**: Balance that minimises total error (MSE = Bias² + Variance)

As model complexity increases: bias typically decreases, variance typically increases. There’s a sweet spot in between.

24.4 Example: Shrinkage Estimators

Consider estimating the mean μ of a normal distribution from n samples $y_i \sim \mathcal{N}(\mu, \sigma^2)$.

Sample mean: $\bar{y} = \frac{1}{n} \sum_i y_i$

- Unbiased: $\mathbb{E}[\bar{y}] = \mu$
- Variance: $\text{Var}(\bar{y}) = \sigma^2/n$
- MSE: σ^2/n (since bias = 0)

Shrinkage estimator: $\tilde{y} = \frac{n}{n+k}\bar{y}$ (shrinks toward zero)

- Biased: $\mathbb{E}[\tilde{y}] = \frac{n}{n+k}\mu \neq \mu$
- Lower variance: $\text{Var}(\tilde{y}) = \left(\frac{n}{n+k}\right)^2 \frac{\sigma^2}{n}$

Bias-Variance Tradeoff in Shrinkage

For the shrinkage estimator with parameter k :

- **Bias:** $\mu - \frac{n}{n+k}\mu = \frac{k}{n+k}\mu$ (increases with k)
- **Variance:** $\left(\frac{n}{n+k}\right)^2 \frac{\sigma^2}{n}$ (decreases with k)

The parameter k controls the compromise:

- $k = 0$: No shrinkage, recover the unbiased sample mean
- Large k : Heavy shrinkage toward zero, low variance but high bias

When $|\mu|$ is small relative to σ/\sqrt{n} (i.e., the true mean is close to our prior belief of zero), the variance reduction can outweigh the bias, giving lower MSE than the unbiased estimator.

This technique is useful when:

- The sample size is small (high variance is a problem)
- There is substantial uncertainty about the sample mean
- We have prior information suggesting the parameter is near zero (or some other value)

Practical Implications

Unbiased is not always best. If an unbiased estimator has high variance, a biased estimator with lower variance may have better overall performance (lower MSE). This insight motivates:

- **Regularisation:** L1 (Lasso), L2 (Ridge) penalties shrink coefficients toward zero
- **Bayesian priors:** Encode beliefs that shrink estimates toward prior mean
- **Ensemble methods:** Averaging multiple models reduces variance
- **Early stopping:** Stop training before the model fully fits the training data

25 Preview: Regularisation and its Geometry

Section Summary

Regularisation adds a penalty to the loss function to control model complexity. L2 (Ridge) shrinks coefficients toward zero; L1 (Lasso) sets some coefficients exactly to zero. Geometrically, regularisation constrains the solution to lie within a region centred at the origin. Full treatment in Week 3.

The bias-variance tradeoff motivates **regularisation**: intentionally introducing bias to reduce variance. We preview the geometric intuition here; detailed treatment follows in Week 3.

25.1 Regularised Loss Functions

Regularised Objectives

Ridge Regression (L2 penalty):

$$\hat{\beta}_{\text{Ridge}} = \arg \min_{\beta} \{ \|y - X\beta\|_2^2 + \lambda \|\beta\|_2^2 \}$$

Lasso (L1 penalty):

$$\hat{\beta}_{\text{Lasso}} = \arg \min_{\beta} \{ \|y - X\beta\|_2^2 + \lambda \|\beta\|_1 \}$$

The hyperparameter $\lambda \geq 0$ controls the strength of regularisation:

- $\lambda = 0$: No regularisation, recover OLS
- $\lambda \rightarrow \infty$: Heavy regularisation, coefficients shrink to zero

Unpacking the penalties:

- $\|\beta\|_2^2 = \sum_j \beta_j^2$ is the squared L2 norm—sum of squared coefficients
- $\|\beta\|_1 = \sum_j |\beta_j|$ is the L1 norm—sum of absolute values

Both penalties discourage large coefficients, but in different ways.

25.2 Geometric Interpretation

The regularised objective can be rewritten as a *constrained* optimisation problem.

Constrained Formulation

By Lagrangian duality, Ridge regression is equivalent to:

$$\hat{\beta}_{\text{Ridge}} = \arg \min_{\beta} \|y - X\beta\|_2^2 \quad \text{subject to} \quad \|\beta\|_2^2 \leq t$$

Similarly for Lasso with an L1 constraint: $\|\beta\|_1 \leq t$.

The Lagrange multiplier λ and constraint bound t are in one-to-one correspondence: larger λ means smaller t .

Geometric picture:

- The OLS solution $\hat{\beta}_{\text{OLS}}$ minimises the RSS. The contours of constant RSS are ellipses (in 2D) or ellipsoids (in higher dimensions) centred at $\hat{\beta}_{\text{OLS}}$.
- The constraint region is a ball (L2: $\|\beta\|_2 \leq t$) or diamond (L1: $\|\beta\|_1 \leq t$) centred at the origin.
- The regularised solution is where the RSS contours first touch the constraint region.

L1 vs L2: Sparsity

Why does L1 give sparse solutions?

The L1 constraint region (a diamond/cross-polytope) has **corners** at the axes. Elliptical contours typically first touch the constraint at a corner, where some coordinates are exactly zero.

The L2 constraint region (a ball) is **smooth**. Contours touch it at a point where typically all coordinates are nonzero (just shrunk).

Rule of thumb:

- **Ridge (L2)**: All coefficients shrunk, none exactly zero—good when all features are relevant
- **Lasso (L1)**: Some coefficients shrunk to exactly zero—performs feature selection

25.3 Ridge Regression: Closed Form

Unlike Lasso, Ridge regression has an analytic solution.

Ridge Solution

$$\hat{\beta}_{\text{Ridge}} = (X^\top X + \lambda I)^{-1} X^\top y$$

Key properties:

- **Always invertible**: Even when $X^\top X$ is singular, $X^\top X + \lambda I$ is invertible for $\lambda > 0$. The λI term ensures positive definiteness.
- **Shrinks toward zero**: As λ increases, $\hat{\beta}_{\text{Ridge}} \rightarrow 0$
- **Bayesian interpretation**: Equivalent to posterior mean with Gaussian prior $\beta \sim \mathcal{N}(0, \sigma^2 / \lambda \cdot I)$

Connection to Bias-Variance

Regularisation trades bias for variance:

- **Bias increases:** We're constraining β away from the OLS solution
- **Variance decreases:** The constraint stabilises estimates against sampling variation
- **Optimal λ :** Chosen via cross-validation to minimise test error

This is the bias-variance tradeoff in action: we accept some systematic error (bias) in exchange for more stable predictions (lower variance).

26 Summary

Key Concepts from Week 2

1. **MLE:** Choose parameters to maximise probability of observed data
2. **NLL:** Negative log-likelihood—the loss function for MLE; minimising NLL = maximising likelihood
3. **i.i.d. assumption:** Independence and identical distribution; substantively meaningful and often violated
4. **Linear regression:** MLE with Gaussian errors \Leftrightarrow least squares; justified by probabilistic assumptions
5. **OLS solution:** $\hat{\beta} = (X^\top X)^{-1} X^\top y$ (orthogonal projection onto column space)
6. **KL divergence:** Measures distribution “distance”; MLE minimises $D_{\text{KL}}(p\|q)$
7. **Variance of $\hat{\beta}$:** Sandwich form $(X^\top X)^{-1} X^\top \mathbb{E}[\epsilon\epsilon^\top] X (X^\top X)^{-1}$; simplifies under homoskedasticity
8. **Robust SEs:** Handle heteroskedasticity via HC estimators
9. **Bayesian inference:** Parameters as random variables; posterior = likelihood \times prior
10. **MAP:** MLE + prior = regularised estimation; Gaussian prior \rightarrow Ridge, Laplace prior \rightarrow Lasso
11. **Convexity:** Guarantees global optimum; enables gradient descent convergence
12. **Gradient descent:** Iterative optimisation; SGD scales to large data
13. **Logistic regression:** Classification via sigmoid; trained with cross-entropy; no closed form
14. **Bias-variance:** $\text{MSE} = \text{Bias}^2 + \text{Variance}$; tradeoff is fundamental
15. **Regularisation:** L1/L2 penalties trade bias for variance; geometric constraint view

ML Lecture Notes: Week 3

High-Dimensional Methods & Regularisation

27 Supervised Learning: A Quick Recap

Section Summary

Supervised learning uses labelled examples to learn a mapping from inputs to outputs. We measure success by comparing predictions to true labels using a loss function. This week, we explore what happens when we expand features to capture nonlinear relationships, and why this expansion creates both opportunities and challenges.

Before diving into high-dimensional methods, let us briefly recall the supervised learning framework. Given inputs X with corresponding labels y , we aim to learn a function f that maps inputs to outputs:

- **Classification:** Learn $f(x) : \mathcal{X} \rightarrow \{0, 1\}$ (or more generally, $\{1, \dots, K\}$ for K classes)
- **Regression:** Learn $f(x) : \mathcal{X} \rightarrow \mathbb{R}$

Performance is measured by some distance or discrepancy between predicted and actual labels: $d(\hat{f}(x), y)$. For example, in OLS regression we use the squared error $(\hat{f}(x) - y)^2$.

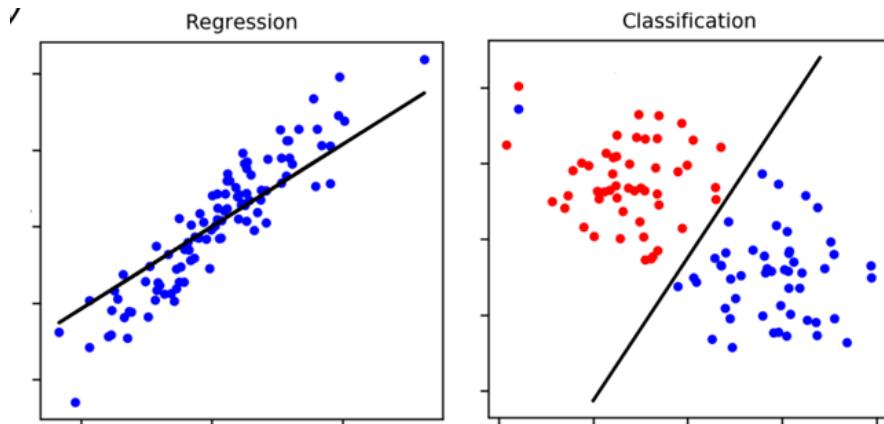


Figure 8: Regression vs classification in supervised learning. Left: regression fits a continuous function through the data. Right: classification finds a decision boundary separating classes.

27.1 OLS Recap

Recall that OLS gives us the optimal linear predictor:

$$\hat{\beta} = (X^\top X)^{-1} X^\top y$$

What this formula says: To find the best linear coefficients $\hat{\beta}$, we:

1. Compute $X^\top X$: This captures how features correlate with each other (the Gram matrix)

2. Invert $(X^\top X)^{-1}$: This “undoes” the feature correlations
3. Multiply by $X^\top y$: This captures how features correlate with the target

The result $\hat{\beta}$ tells us the independent contribution of each feature to predicting y , accounting for correlations between features.

We then plug these coefficients back to get predictions:

$$\hat{y} = X\hat{\beta}$$

In general, we assume the first column of X is a column of ones (the intercept), giving us a matrix of dimension $n \times (p + 1)$. The prediction is then:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 X_1 + \cdots + \hat{\beta}_p X_p$$

But what if the true relationship is nonlinear? One approach is **feature expansion**.

28 From Linear to Nonlinear: Polynomial Regression

Section Summary

Feature expansion transforms simple linear models into powerful nonlinear approximators while preserving the tractability of OLS. The key insight: “linear regression” means linear in *parameters*, not features. However, high-degree polynomials introduce numerical instability (ill-conditioned matrices) and edge oscillations (Runge’s phenomenon). These problems motivate regularisation and alternative basis functions.

OLS gives us the optimal linear predictor:

$$\hat{\beta} = (X^\top X)^{-1} X^\top y \Rightarrow \hat{y} = X\hat{\beta}$$

But what if the true relationship is nonlinear? One approach: **feature expansion**.

Polynomial Regression

For a single feature x , expand to polynomial basis:

$$f(x) = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_M x^M$$

This is still **linear in parameters** β —we’re just using transformed features $[1, x, x^2, \dots, x^M]$.

The design matrix becomes:

$$X = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^M \\ 1 & x_2 & x_2^2 & \cdots & x_2^M \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^M \end{bmatrix}$$

Unpacking the polynomial design matrix:

- Each row corresponds to one observation x_i
- Each column corresponds to a power of x : the j -th column contains x_i^{j-1} for all observations
- The matrix has n rows (observations) and $M + 1$ columns (including the intercept)

- We can apply standard OLS to this matrix, treating each power as a separate “feature”

Key Insight: “Linear” Refers to Parameters

“Linear regression” means linear in **parameters**, not in features. We can model arbitrarily complex relationships by transforming features—polynomials, interactions, logarithms, etc.—while still using the OLS machinery.

The polynomial order M is our **measure of model complexity**: it determines how well the model can fit the training data.

28.1 Why Polynomials Are Attractive (In Theory)

Polynomials can represent a huge class of functions, making them highly flexible and mathematically convenient:

Theoretical Foundations of Polynomial Approximation

- **Taylor series:** Any smooth (infinitely differentiable) function can be approximated by its Taylor polynomial around any point. For a function f expanded around a :

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!} (x - a)^k$$

Truncating this series gives a polynomial approximation.

- **Weierstrass approximation theorem:** Any continuous function on a closed interval $[a, b]$ can be uniformly approximated by polynomials to arbitrary precision. Formally: for any $\epsilon > 0$, there exists a polynomial $p(x)$ such that $|f(x) - p(x)| < \epsilon$ for all $x \in [a, b]$.

These results guarantee that polynomials are “universal approximators” for continuous functions—given enough terms, they can approximate any continuous function arbitrarily well.

Why this matters for machine learning: These theoretical results suggest that if we use high-enough degree polynomials, we should be able to fit any smooth relationship in our data. However, “can approximate” does not mean “easy to estimate from finite data”—this gap between theoretical expressiveness and practical learnability is central to understanding overfitting.

28.2 Feature Expansion: Power and Peril

Feature expansion gives us flexibility, but at a cost. Understanding this tradeoff is essential for effective model building.

The Expansion Explosion

Starting with p original features, consider common expansions:

- **Polynomial degree M :** For a single feature, we get $M + 1$ terms. For p features with all interactions up to degree M , we get $\binom{p+M}{M}$ terms.
- **Pairwise interactions:** Adding $x_i x_j$ terms gives $\binom{p}{2} = \frac{p(p-1)}{2}$ new features.
- **Degree-2 polynomial with interactions:** Grows as $O(p^2)$.

Example: With $p = 100$ original features:

- Pairwise interactions: 4,950 additional features
- Full degree-2: 5,151 total features
- Full degree-3: 176,851 total features

Unpacking the combinatorics: The formula $\binom{p+M}{M}$ counts the number of ways to distribute M units of “degree” among p features (including the option of assigning all to one feature). For example, with $p = 2$ features (x_1, x_2) and $M = 2$:

- Degree 0: constant (1 term)
- Degree 1: x_1, x_2 (2 terms)
- Degree 2: $x_1^2, x_1 x_2, x_2^2$ (3 terms)

Total: $\binom{2+2}{2} = 6$ terms.

Feature Expansion Tradeoffs

Benefits	Costs
Captures nonlinear relationships	Increases effective dimension
Can approximate any smooth function	Triggers curse of dimensionality
Remains tractable (OLS still applies)	Ill-conditioned design matrix
No need to specify functional form	Overfitting risk increases

When to expand features:

- You have strong reason to believe the relationship is nonlinear
- You have enough data to support the additional parameters
- You will use regularisation to control complexity
- The domain suggests specific transformations (e.g., log-income, interaction between treatment and subgroup)

When to be cautious:

- n is small relative to p
- You’re already seeing signs of overfitting
- Interpretability is important (expanded models are harder to explain)
- The original features are already correlated (expansion worsens multicollinearity)

28.3 The Problem: Choosing M

Higher-degree polynomials are more expressive but risk overfitting:

- $M = 1$: Straight line (may underfit)
- $M = 3$: Cubic (often reasonable)
- $M = 15$: Wiggly curve that passes through every training point (overfits)

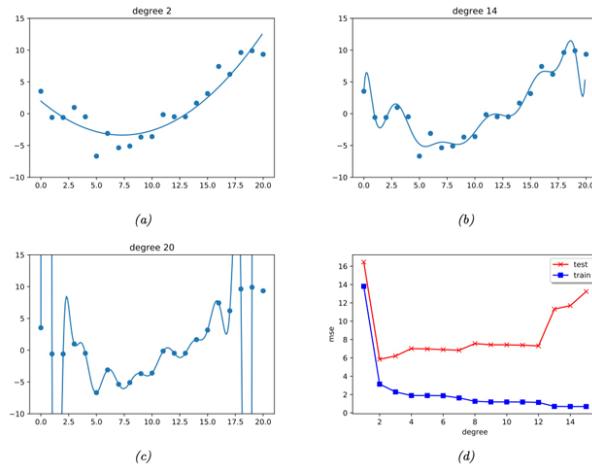


Figure 9: Effect of polynomial degree on fit. Higher degrees fit training data perfectly but generalise poorly. This illustrates the fundamental tension in model selection: low M underfits (misses the pattern), high M overfits (memorises noise).

28.4 Numerical Instability in High-Degree Polynomials

NB!

[Polynomials Are Global Approximators] Polynomials are **global approximators**— changing the polynomial anywhere affects it everywhere. This leads to fundamental problems with **edges** and **variance**:

1. **Runge's phenomenon**: High-degree polynomials oscillate wildly near boundaries when interpolating, even for smooth underlying functions
2. **High variance at edges**: As polynomial degree increases, behaviour becomes increasingly erratic near domain boundaries
3. **Sensitivity to data**: Small changes in data points can cause radically large differences in predictions throughout the entire domain

Because polynomials are global approximators, changes to improve the fit in one part of the domain can have far-reaching effects throughout the entire domain, including unwanted oscillations at the edges.

For polynomials of large degrees (x^k where k is large), two main issues contribute to **numerical instability**:

Sources of Numerical Instability

- Magnitude of polynomial terms:** As the degree k increases, x^k grows rapidly for $|x| > 1$. This leads to extremely large values that are difficult to manage computationally.

Example: For $x = 2$ and $k = 20$, we have $x^k = 2^{20} \approx 10^6$. For $k = 50$, $x^k \approx 10^{15}$.

- Magnitude of coefficients:** To compensate for large polynomial terms, the fitting process produces very small (or very large) coefficients β_k . These coefficients must scale the polynomial terms back to fit the data.

What happens: As x^k explodes, β_k must shrink correspondingly to keep the fit reasonable.

The combination of very large polynomial terms and small coefficients leads to numerical instability: small changes in data or coefficients produce disproportionately large changes in predictions.

Why this is problematic: When we compute $\beta_k \cdot x^k$ where $\beta_k \approx 10^{-10}$ and $x^k \approx 10^{10}$, we're multiplying a very small number by a very large number. Floating-point arithmetic has limited precision (about 16 significant digits in double precision), so this operation loses accuracy. The result might be correct to only a few significant figures, making predictions unreliable.

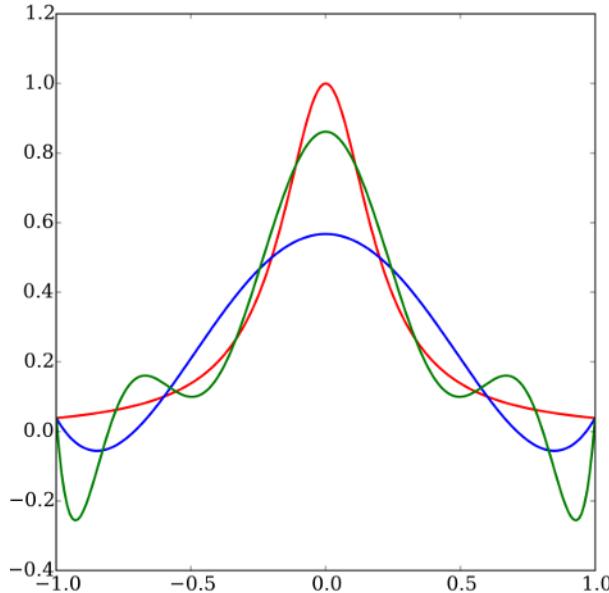


Figure 10: Numerical instability in high-degree polynomials: small perturbations in data cause large changes in the fitted curve, particularly near the edges of the domain.

28.4.1 Condition Numbers: A Deeper Look

The **condition number** formalises the notion of numerical stability. It tells us how sensitive a computation is to small perturbations in the input.

Condition Number: Formal Definition

The **condition number** of an invertible matrix A measures how much the solution $x = A^{-1}b$ changes when we perturb b . Formally, for any matrix norm:

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

Key property: If we perturb b to $b + \delta b$, the relative change in the solution satisfies:

$$\frac{\|x - \tilde{x}\|}{\|x\|} \leq \kappa(A) \cdot \frac{\|\delta b\|}{\|b\|}$$

The condition number bounds how much *relative error* in the input gets amplified in the output.

Unpacking the definition:

- $\|A\|$ measures how much A can “stretch” a vector
- $\|A^{-1}\|$ measures how much A^{-1} can “stretch” a vector
- Their product $\kappa(A)$ measures the worst-case amplification of errors through the system $Ax = b$

Intuition: Think of $\kappa(A)$ as an “error amplification factor.” If $\kappa(A) = 10^6$ and your input has 10^{-8} relative error (typical for double precision floating point), your output could have 10^{-2} relative error—two correct decimal places at best.

For symmetric positive definite matrices (like $X^\top X$ in OLS), the condition number simplifies:

$$\kappa(A) = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)}$$

Why this form? For symmetric matrices, the matrix norm equals the largest eigenvalue, and the inverse’s norm equals the reciprocal of the smallest eigenvalue. The condition number thus measures the “spread” of eigenvalues—how elongated the matrix’s action is in different directions.

Condition Number Interpretation

- $\kappa \approx 1$: Well-conditioned, numerically stable
- $\kappa \sim 10^3$: Moderate—expect to lose about 3 digits of precision
- $\kappa \sim 10^8$: Ill-conditioned—results may be meaningless in double precision
- $\kappa = \infty$: Singular matrix, non-invertible

Rule of thumb: In double precision (about 16 digits), you lose roughly $\log_{10}(\kappa)$ digits of precision. If $\kappa \approx 10^{16}$, you have essentially no reliable digits left.

Connection to near-singularity: A matrix is ill-conditioned when it is “almost singular”—its smallest eigenvalue is tiny relative to the largest. Geometrically, the matrix stretches space enormously in some directions but barely at all in others, making inversion numerically treacherous.

Worked Example: Vandermonde Matrix Conditioning

Consider fitting a polynomial of degree M to n points x_1, \dots, x_n uniformly spaced in $[0, 1]$. The design matrix is the **Vandermonde matrix**:

$$V = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^M \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^M \end{bmatrix}$$

For $n = 20$ uniformly spaced points, the condition number of $V^\top V$ grows rapidly:

Degree M	$\kappa(V^\top V)$	Effective precision
5	$\sim 10^4$	~ 12 digits
10	$\sim 10^{10}$	~ 6 digits
15	$\sim 10^{16}$	Nearly singular
20	$> 10^{20}$	Numerically singular

Consequence: With $M = 15$, small changes in y (e.g., rounding errors) cause wild swings in $\hat{\beta}$. The OLS solution becomes meaningless despite the matrix being technically invertible.

Why does the condition number grow so fast? The columns of the Vandermonde matrix become increasingly similar as the degree grows. The column $[x_1^{10}, x_2^{10}, \dots, x_n^{10}]^\top$ is very similar to $[x_1^{11}, x_2^{11}, \dots, x_n^{11}]^\top$ because raising to a slightly higher power only slightly changes the relative values. This near-collinearity makes $X^\top X$ nearly singular.

NB!

When $\kappa(X^\top X) \approx 10^{16}$ (machine precision), numerical solvers may return garbage. Always check the condition number before trusting OLS results with high-degree polynomials or correlated features!

28.5 Runge's Phenomenon

Even when we can solve the numerical equations exactly, high-degree polynomial interpolation can fail spectacularly.

Runge's Phenomenon

When interpolating the function $f(x) = \frac{1}{1+25x^2}$ on $[-1, 1]$ using a polynomial of degree n through $n + 1$ equally spaced points, the interpolation error **diverges** as $n \rightarrow \infty$:

$$\max_{x \in [-1, 1]} |f(x) - p_n(x)| \rightarrow \infty \quad \text{as } n \rightarrow \infty$$

The polynomial fits well in the centre but oscillates wildly near the boundaries.

Why does this happen? Polynomials are **global approximators**—changing the polynomial anywhere affects it everywhere. With equally spaced nodes, the polynomial must “work harder” near the edges to pass through the interpolation points, causing oscillations that grow with degree.

A more detailed explanation: The function $f(x) = \frac{1}{1+25x^2}$ is smooth everywhere on the real

line, but it has singularities in the complex plane at $x = \pm i/5$. These complex singularities, though invisible on the real line, limit how well polynomials can approximate the function. The approximation struggles most at the endpoints because that's where the “pull” from the complex singularities is strongest relative to the constraining data points.

Connection to overfitting: Runge's phenomenon is a deterministic analogue of overfitting. The polynomial exactly matches the data points (zero training error) but performs terribly between them (high “test” error). The oscillations are most severe at the boundaries—precisely where we have the least data to constrain the fit.

Runge's Phenomenon: Key Insights

- High-degree polynomials on equally spaced points can **diverge** rather than converge
- Oscillations are worst at the **boundaries** of the domain
- More data points (with equal spacing) makes things **worse**, not better
- This is fundamentally about the **global** nature of polynomial approximation

28.5.1 Solutions to Runge's Phenomenon

1. **Chebyshev nodes:** Instead of equally spaced points, use nodes clustered near the boundaries:

$$x_k = \cos\left(\frac{2k-1}{2n}\pi\right), \quad k = 1, \dots, n$$

Chebyshev nodes minimise the maximum interpolation error. With these nodes, polynomial interpolation *converges* for smooth functions.

Why Chebyshev nodes work: They place more points near the boundaries where interpolation error tends to be largest. This gives the polynomial more “anchor points” in the problematic regions.

2. **Regularisation:** Rather than interpolating (passing through all points exactly), fit a *regularised* polynomial that trades off data fidelity against smoothness. This prevents the wild oscillations that occur when forcing exact fit.

3. **Local methods (splines):** Use **piecewise polynomials** that are only responsible for fitting a local region. Cubic splines, for instance, use degree-3 polynomials between each pair of knots, joined smoothly. Changes in one region don't propagate globally.

4. **Truncate polynomial degree:** Accept that beyond a certain degree, adding more terms hurts rather than helps. Use cross-validation to select the optimal degree.

NB!

Runge's phenomenon warns: **more flexibility is not always better**. A model that perfectly fits training data may be useless for prediction. This motivates the regularisation techniques we develop in Section 31.

Alternatives to High-Degree Polynomials

Because polynomials are global approximators, trying to fit functions with sharp edges or rapid changes leads to high variance and oscillatory behaviour at domain boundaries. This motivates alternative approaches:

- **Splines**: Piecewise polynomials providing *local* approximation
- **Regularisation**: Penalising complexity to control oscillations
- **Kernel methods**: Implicit feature expansion without explicit polynomial terms

29 The Curse of Dimensionality

Section Summary

High-dimensional spaces behave counterintuitively: volume concentrates in thin shells near the boundary, distances between points become nearly uniform, and local methods fail because “local” neighbourhoods contain almost no data. These phenomena explain why flexible methods that work in low dimensions break down as p grows, and why dimension reduction and regularisation become essential.

When we expand features (e.g., polynomial terms, interactions), the effective dimension of our problem grows rapidly. This triggers a suite of counterintuitive behaviours collectively known as the **curse of dimensionality**.

The term was coined by Richard Bellman in the context of dynamic programming, where the computational cost grows exponentially with the number of state variables. In machine learning, the curse manifests as a fundamental limitation on what can be learned from finite data in high dimensions.

29.1 Volume Concentration

Our low-dimensional intuitions about space fail dramatically in high dimensions.

Volume of High-Dimensional Spheres

Consider a unit hypercube $[0, 1]^p$ in p dimensions. What fraction of its volume lies within a ball of radius r centred at the origin?

The volume of a p -dimensional ball of radius r is:

$$V_p(r) = \frac{\pi^{p/2}}{\Gamma(p/2 + 1)} r^p$$

As $p \rightarrow \infty$, this volume becomes negligible compared to the hypercube. Almost all the volume of the cube lies in the “corners”—far from the centre.

Unpacking the formula:

- $\Gamma(\cdot)$ is the gamma function, a generalisation of factorial: $\Gamma(n + 1) = n!$ for integers
- The key insight is the r^p factor: even for r close to 1, raising it to a high power makes it tiny

- For $r = 0.9$ and $p = 100$: $0.9^{100} \approx 2.7 \times 10^{-5}$

Consequence: In high dimensions, data points are almost always near the boundary of any bounded region. There is essentially no “interior.”

Shell Concentration

Consider a unit ball in p dimensions. What fraction of its volume lies in a thin shell between radius $1 - \epsilon$ and 1?

$$\frac{V_p(1) - V_p(1 - \epsilon)}{V_p(1)} = 1 - (1 - \epsilon)^p \rightarrow 1 \quad \text{as } p \rightarrow \infty$$

For $p = 100$ and $\epsilon = 0.1$: the shell contains over 99.997% of the volume.

Derivation: Since $V_p(r) \propto r^p$, we have:

$$\frac{V_p(1) - V_p(1 - \epsilon)}{V_p(1)} = \frac{1^p - (1 - \epsilon)^p}{1^p} = 1 - (1 - \epsilon)^p$$

For small ϵ and large p , use the approximation $(1 - \epsilon)^p \approx e^{-\epsilon p}$, which goes to 0 rapidly as p increases.

Shell Concentration Intuition

In high dimensions, essentially **all volume is near the surface**. If you sample uniformly from a high-dimensional ball, almost every point will be close to the boundary. The “centre” of the distribution contains virtually no probability mass.

Analogy: Imagine an orange in 3D—the peel is thin relative to the fruit. Now imagine a 100-dimensional orange: the “peel” (outer shell) contains essentially all the volume, and the “fruit” (interior) is negligible.

29.2 Distance Concentration

Uniform Distance Phenomenon

Let X_1, \dots, X_n be i.i.d. points uniformly distributed in $[0, 1]^p$. As $p \rightarrow \infty$:

$$\frac{\max_i \|X_i - X_j\|}{\min_i \|X_i - X_j\|} \rightarrow 1$$

All pairwise distances become nearly equal—the notions of “near” and “far” become meaningless.

Intuition: In high dimensions, the expected squared distance between two random points is:

$$\mathbb{E}[\|X - Y\|^2] = \sum_{j=1}^p \mathbb{E}[(X_j - Y_j)^2] = p \cdot \mathbb{E}[(X_1 - Y_1)^2]$$

By the law of large numbers, the average of p independent terms concentrates around its expectation. Hence all distances cluster near $\sqrt{p \cdot \text{const.}}$

More precisely: The variance of $\|X - Y\|^2$ grows as $O(p)$, but its mean grows as $O(p)$. So the coefficient of variation (standard deviation divided by mean) decreases as $O(1/\sqrt{p})$. This means distances become increasingly concentrated around their mean.

29.3 Implications for Machine Learning

NB!

Local methods (k-nearest neighbours, kernel regression, local polynomial regression) assume that nearby points behave similarly. In high dimensions:

1. The “neighbourhood” needed to contain k points grows to encompass most of the space
2. All points are approximately equidistant, so “nearest” neighbours aren’t meaningfully near
3. Accurate local estimation would require exponentially many points: $n \propto k^p$

Sample Size Requirements

To maintain a fixed neighbourhood size r that captures a proportion f of the data:

$$r \propto f^{1/p}$$

As p grows, the radius must approach the diameter of the space. To maintain genuine locality, you need sample sizes that grow **exponentially** in p .

Worked example: Suppose we want to capture 1% of the data within our neighbourhood. The required radius is:

- In $p = 2$ dimensions: $r = 0.01^{1/2} = 0.1$ (10% of the range in each dimension)
- In $p = 10$ dimensions: $r = 0.01^{1/10} = 0.63$ (63% of the range in each dimension)
- In $p = 100$ dimensions: $r = 0.01^{1/100} = 0.955$ (95.5% of the range in each dimension)

In 100 dimensions, to capture even 1% of the data, we need a “neighbourhood” that spans 95% of the space in every direction—this is not local at all!

This is why regularisation, dimension reduction, and structured models become essential in high dimensions: we cannot rely on local averaging when every point is isolated in a vast empty space.

Escaping the Curse

1. **Regularisation:** Impose structure (smoothness, sparsity) to reduce effective complexity
2. **Dimension reduction:** PCA, feature selection, or learned embeddings
3. **Structured models:** Linear models, additive models, neural networks with parameter sharing
4. **Domain knowledge:** Use problem structure to identify relevant low-dimensional subspaces

The curse of dimensionality is not a death sentence—it’s a call to be smarter about how we model high-dimensional data. Real-world data often lies on or near low-dimensional manifolds, making learning possible despite the nominal dimension being high.

30 Decomposing Prediction Error

Section Summary

Prediction error decomposes into **approximation error** (how well the best function in our hypothesis class matches the truth) and **estimation error** (how well we can find that best function from finite data). Simple models have high approximation but low estimation error; complex models have the reverse. This formalises the bias-variance tradeoff and motivates regularisation as a principled complexity control.

To understand model selection, we need to formalise what we're trying to minimise. This section develops the theoretical framework for understanding generalisation.

30.1 The Bias-Variance Tradeoff: A First Look

Before diving into the formal framework, let us recall the fundamental decomposition:

Bias-Variance Decomposition

For an estimator $\hat{\theta}$ of a parameter θ :

$$\text{bias}(\hat{\theta}) = \mathbb{E}[\hat{\theta}] - \theta$$

$$\text{MSE}(\hat{\theta}) = \text{Var}(\hat{\theta}) + \text{bias}(\hat{\theta})^2$$

Unpacking the decomposition:

- **Bias**: The systematic error—how far, on average, is our estimator from the truth?
- **Variance**: The random error—how much does our estimator vary across different samples?
- **MSE**: The total expected squared error, which combines both sources

As model complexity increases:

- **Bias decreases**: The function can “express” the data better—a more flexible model can capture the true underlying pattern
- **Variance increases**: The function becomes harder to estimate reliably—more parameters mean more sensitivity to the particular sample

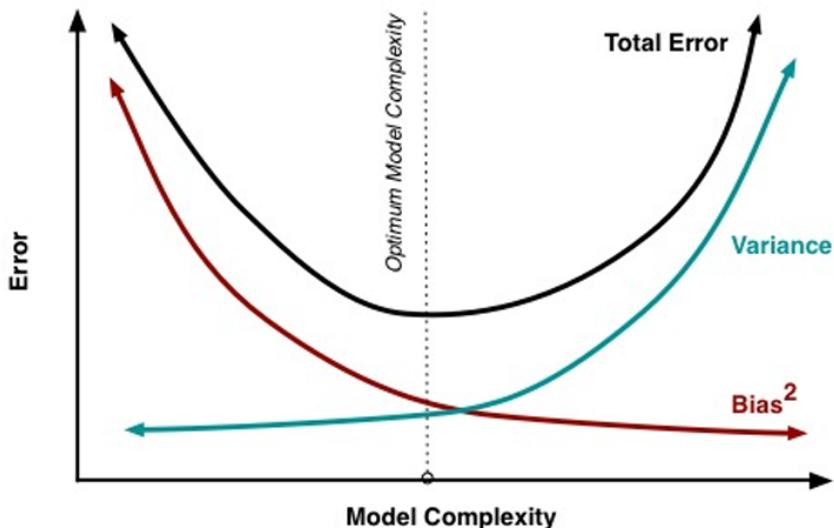


Figure 11: Bias-variance tradeoff: as model complexity increases, bias decreases but variance increases. The optimal complexity minimises total error (MSE). This U-shaped curve is fundamental to model selection.

30.2 Evaluation Metrics: Defining “Risk”

Different metrics capture different aspects of model performance. The choice of metric defines what “risk” we’re trying to minimise.

Key Concepts

1. **Model:** How you predict $f(x)$ from X
2. **Loss function:** Quantifies the discrepancy between actual outcome y and predicted outcome $f(x)$:

$$\ell(y, f(x))$$
3. **Risk:** The expected loss of a model—the metric we minimise in ERM

“Risk” is a generalised concept referring to the **expected loss or error of a model with respect to its predictions on new data**. It quantifies how much, on average, the model’s predictions deviate from actual values according to the chosen loss function.

Common Evaluation Metrics

For Classification:

- **Accuracy:** Proportion of correct predictions. Use when false positives and false negatives have similar costs.
- **Precision:** $\frac{TP}{TP+FP}$. Use when the cost of false positives is high (e.g., spam filtering—you don't want legitimate emails marked as spam).
- **Recall:** $\frac{TP}{TP+FN}$. Use when the cost of false negatives is high (e.g., disease screening—you don't want to miss actual cases).
- **AUC-ROC:** Area under the ROC curve. Summarises performance across all classification thresholds; higher is better. Risk could be considered as $1 - \text{AUC}$.

For Regression:

- **MSE:** Mean squared error. Penalises large errors heavily. Directly quantifies risk as expected squared error.
- **MAE:** Mean absolute error. More robust to outliers.
- **R^2 :** Proportion of variance explained.

Unpacking the classification metrics:

- TP (True Positives): Correctly predicted positive cases
- FP (False Positives): Incorrectly predicted positive (actually negative)—Type I error
- FN (False Negatives): Incorrectly predicted negative (actually positive)—Type II error
- TN (True Negatives): Correctly predicted negative cases

		True	
		0	1
Predicted	0	✓	Type II
	1	Type I	✓

Figure 12: Confusion matrix showing Type I errors (false positives) and Type II errors (false negatives). These frame the loss in terms of incorrect predictions.

NB!

[Accuracy Can Be Misleading] In imbalanced datasets, accuracy can be deceptive. A classifier that always predicts the majority class achieves high accuracy but is useless.

Example: If 99% of emails are not spam, a classifier that predicts “not spam” for everything achieves 99% accuracy but catches zero spam. Precision and recall provide more insight in such cases.

30.3 Population Risk vs Empirical Risk

Population Risk (True Risk)

$$R(f) = R_{f,p^*} = \mathbb{E}_{(x,y) \sim p^*} [\ell(y, f(x))]$$

The expected loss over the **true data distribution** p^* . This is what we ultimately care about, but we cannot compute it—we don't know p^* .

Unpacking the notation:

- R_{f,p^*} or $R(f)$: Population risk for model f
- $\mathbb{E}_{p^*}[\cdot]$: Expectation over the true population distribution
- $\ell(y, f(x))$: Loss function measuring discrepancy between true y and predicted $f(x)$

Population risk is the gold standard for model performance: it indicates how well the model would perform in general, beyond just the observed data. But since the true distribution p^* is unknown, direct computation is infeasible.

What this formula means in practice: If we could somehow draw infinitely many samples from the true distribution and evaluate our model on each one, population risk is the average loss we would observe. It represents the model's “true” performance on the task.

Empirical Risk

$$\hat{R}(f) = \frac{1}{n} \sum_{i=1}^n \ell(y_i, f(x_i))$$

The average loss over our **training sample**. This we can compute, but it's only an estimate of population risk.

The empirical risk is based on the **empirical distribution** of the sample data, which approximates the true underlying distribution.

The gap between these two: Empirical risk uses only the n samples we have; population risk averages over all possible samples. As $n \rightarrow \infty$, empirical risk converges to population risk (by the law of large numbers), but for finite n there's always a gap—and this gap is precisely what causes overfitting.

Empirical Risk Minimisation (ERM)

$$\hat{f}_{\text{ERM}} = \arg \min_{f \in \mathcal{H}} \hat{R}(f) = \arg \min_{f \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^n \ell(y_i, f(x_i))$$

Find the function in hypothesis class \mathcal{H} that minimises training loss. This is the foundation of most ML algorithms.

Components:

- \hat{f}_{ERM} : The model we seek
- $\arg \min_{f \in \mathcal{H}}$: Search over hypothesis space \mathcal{H}
- The sum: Empirical risk (average loss on training data)

ERM seeks to approximate optimal population risk by minimising loss on the observed dataset.

NB!

[The Problem with Pure ERM] If you choose a model based on ERM alone, you will overfit to the training data and end up with a high-order polynomial (or similarly complex model)—which is problematic!

Pure ERM drives training loss toward zero while test loss remains high. We need additional considerations...

30.4 Three Levels of Optimality

To understand what we can and cannot achieve, we distinguish three levels of model quality:

Hierarchy of Functions

1. **Bayes optimal:** $f^{**} = \arg \min_f R(f)$
 - Best possible function over *all* functions
 - Theoretical ideal; typically unachievable
 - Minimises true risk $R(f)$ without any constraints
 - This is an ideal function—perhaps more complex than any polynomial, perhaps discontinuous
 - Can never be observed
2. **Best in class:** $f^* = \arg \min_{f \in \mathcal{H}} R(f)$
 - Best function within our hypothesis class \mathcal{H}
 - Still uses true risk (unknown in practice)
 - Represents the best we could achieve given our modelling choice
 - We cannot reach f^{**} if the true function is not in \mathcal{H}
3. **Empirical best:** $\hat{f}_n = \arg \min_{f \in \mathcal{H}} \hat{R}(f)$
 - Best function based on training data (minimises empirical risk)
 - What we actually compute
 - Trained on n samples—a subset of the full population
 - Aims to approximate f^* by minimising observed errors

The progression: We want f^{**} (the ultimate goal), we settle for f^* (the best our model class can do), and we actually compute \hat{f}_n (what we can find from data). Each step introduces error.

30.5 Approximation vs Estimation Error

The gap between what we achieve and the theoretical best decomposes into two sources. This decomposition is fundamental to understanding model selection.

Error Decomposition

$$\underbrace{R(\hat{f}_n) - R(f^{**})}_{\text{Total excess risk}} = \underbrace{R(f^*) - R(f^{**})}_{\text{Approximation error}} + \underbrace{R(\hat{f}_n) - R(f^*)}_{\text{Estimation error}}$$

Or equivalently, thinking of this as $R_3 - R_1 = (R_2 - R_1) + (R_3 - R_2)$.

Unpacking the decomposition:

- **Total excess risk:** How much worse is our learned model compared to the best possible?
- **Approximation error:** How much worse is the best model in our class compared to the best possible?
- **Estimation error:** How much worse is our learned model compared to the best in our class?

30.5.1 Approximation Error

Approximation Error (also called: bias, model misspecification):

- Gap between Bayes optimal (f^{**}) and best-in-class (f^*)
- Due to **limitations of our hypothesis class**
- Does **not** decrease with more data
- Reduced by using more expressive model classes
- Quantifies how well the best theoretical model in our chosen hypothesis space can approximate the true best model
- This error is **theory-based**—inherent to our modelling choice

Approximation Error: The Cost of Our Modelling Choice

Approximation error measures how much worse our chosen model class is compared to the best possible. We can never eradicate this entirely; we can only do a better job of selecting our function class. We will always pay some cost based on our modelling choice.

Example: If the true relationship is a sigmoid but we only consider linear functions, no amount of data will help—our best linear fit will always have positive approximation error.

30.5.2 Estimation Error

Estimation Error (also called: variance, generalisation error):

- Gap between best-in-class (f^*) and what we actually learn (\hat{f}_n)
- Due to **finite training data**
- **Decreases** with more data (typically $O(1/\sqrt{n})$)
- **Increases** with model complexity (overfitting)
- This error is **empirically-based**—from estimating from finite samples

NB!

[Estimation Error: What We Can Control] The estimation error is influenced by:

1. **Sample size:** Decreases as n increases
2. **Model complexity:** Increases if complexity is too high relative to available data (overfitting)

This is where we *can* do something—unlike approximation error, which is fixed by our model choice.

30.5.3 The Fundamental Tradeoff

Approximation vs Estimation: The Core Tradeoff

	Approximation Error	Estimation Error
Simple model	High	Low
Complex model	Low	High

More expressive models reduce approximation error but increase estimation error. The optimal model balances these.

This is analogous to the bias-variance tradeoff: we could make \mathcal{H} a huge class of functions, but this increases complexity and makes estimation harder. We want to choose a model that balances the tradeoff between approximation and estimation errors.

Balancing the Errors

The total difference in risk between the theoretical best possible model and our empirically best model can be understood through two fundamental challenges:

1. **Choosing the right model class** (approximation error)
2. **Accurately estimating the best model within that class from limited data** (estimation error)

Minimising total error involves balancing these two sources. Improving the model class to reduce approximation error might increase model complexity, potentially increasing estimation error if additional data is not available.

30.6 Estimating Generalisation Error

We cannot compute true risk, but we can **estimate** generalisation performance using held-out data.

Train-Test Split

Partition data into:

- **Training data** $p_{\text{train}}(x, y)$: Do ERM—minimise loss on these points, learning the underlying pattern
- **Testing data** $p_{\text{test}}(x, y)$: Evaluate model performance—specifically, its ability to generalise to new, unseen data

By evaluating on testing data, we can estimate generalisation error:

$$\underbrace{\mathbb{E}_{p^*} R(\hat{f}_n) - R(f^*)}_{\text{Estimation/Generalisation Error}} \approx \underbrace{\mathbb{E}_{p_{\text{test}}} [\ell(y, \hat{f}_n)]}_{\text{Test Loss}} - \underbrace{\mathbb{E}_{p_{\text{train}}} [\ell(y, \hat{f}_n)]}_{\text{Training Loss}}$$

We are comparing:

- **Training Loss**: How well we *thought* we did—average loss on the training dataset
- **Test Loss**: How well we *actually* did—average loss on unseen data

Generalisation gap:

$$\text{Gap} = \hat{R}_{\text{test}}(\hat{f}_n) - \hat{R}_{\text{train}}(\hat{f}_n)$$

A large gap indicates overfitting: the model performs much better on training data than on new data.

NB!

[The Optimism of Pure ERM] Generalisation/estimation error quantifies how **overly optimistic** we were when using pure ERM.

In an overfitted model:

- Approximation error is basically zero (the model can express the training data perfectly)
- But estimation/generalisation error is high (the gap between training loss approaching zero and test loss remaining high is large)

ERM makes us overly optimistic because we are overfitting to the data—it will reduce training loss to zero but this does not translate to good test performance.

Key Takeaways on Generalisation

- **Generalisation gap** = difference between test and training performance. Small gap indicates good generalisation; large gap suggests overfitting.
- Since we cannot directly observe true expected risk over the entire distribution p^* , we use train/test splits to estimate how well our model will perform in practice.
- The testing data provides an estimate—the model’s true performance could vary in completely new contexts.

31 Regularisation

Section Summary

Regularisation penalises model complexity to prevent overfitting. **Ridge** (L2) shrinks coefficients toward zero with a closed-form solution and guaranteed invertibility. **Lasso** (L1) produces sparse solutions, setting some coefficients exactly to zero. **Elastic net** combines both. These can be understood through multiple lenses: necessity (invertibility), bias-variance tradeoff, Bayesian priors, and geometry.

Regularisation adds a penalty for model complexity, trading off fit against simplicity. It prevents models from overfitting by introducing additional information or constraints to discourage overly complex models.

31.1 The Mechanics of Regularisation

Overfitting occurs when a model learns patterns specific to the training data, including noise, to the extent that it performs poorly on new data. Regularisation addresses this by adding a **penalty on the size of model parameters** to the loss function.

Regularised Objective

$$\mathcal{L}(\theta; \lambda) = \underbrace{\frac{1}{n} \sum_{i=1}^n \ell(y_i, f(x_i; \theta))}_{\text{Loss (data fit)}} + \lambda \underbrace{C(\theta)}_{\text{Complexity penalty}}$$

where $\lambda \geq 0$ controls the regularisation strength:

- $\lambda = 0$: No regularisation (pure ERM)
- $\lambda \rightarrow \infty$: Ignore data, minimise complexity only

The parameter λ manages the tradeoff between fitting the data and keeping the model simple.

Unpacking the objective:

- The first term (loss) pulls parameters toward values that fit the data well
- The second term (penalty) pulls parameters toward “simple” values (often zero)
- λ controls which pull is stronger
- The optimal θ balances these competing objectives

31.2 Uses of Regularisation

1. **Prevent overfitting**: By penalising large coefficients, regularisation reduces model complexity, leading to lower variance and less overfitting
2. **Improve generalisation**: A simpler model with smaller coefficients is less sensitive to noise in training data, making it better at predicting outcomes for unseen data
3. **Feature selection (L1)**: By driving some coefficients to zero, L1 regularisation helps identify the most important features

31.3 Ridge Regression (L2 Regularisation)

Ridge Regression

$$\mathcal{L}_{\text{ridge}}(\beta; \lambda) = \frac{1}{n} \|y - X\beta\|_2^2 + \lambda \|\beta\|_2^2$$

where $\|\beta\|_2^2 = \beta^\top \beta = \sum_j \beta_j^2$ penalises the **squared magnitude** of coefficients.

Closed-form solution:

$$\hat{\beta}_{\text{ridge}} = (X^\top X + \lambda I_p)^{-1} X^\top y$$

Note: When $\lambda = 0$, this reduces to the standard OLS solution.

Deriving the Ridge solution:

Taking the gradient of the Ridge objective and setting it to zero:

$$\begin{aligned}\nabla_{\beta} \mathcal{L} &= -\frac{2}{n} X^T (y - X\beta) + 2\lambda\beta = 0 \\ X^T X\beta + n\lambda\beta &= X^T y \\ (X^T X + n\lambda I)\beta &= X^T y\end{aligned}$$

The factor of n is often absorbed into λ (different conventions exist), giving the stated solution.

Why Ridge Works

1. **Shrinkage:** Coefficients are pulled toward zero, reducing variance
2. **Guaranteed invertibility:** Adding λI ensures $X^T X + \lambda I$ is always invertible
3. **Stabilises conditioning:** Increases smallest eigenvalues, reducing κ

31.3.1 Ridge as Rescaled OLS

For orthonormal X (i.e., $X^T X = I$, where each column is independent and normalised):

$$\hat{\beta}_{\text{ridge}} = (I + \lambda I)^{-1} X^T y = \frac{1}{1 + \lambda} X^T y = \frac{\hat{\beta}_{\text{OLS}}}{1 + \lambda}$$

Ridge uniformly shrinks all coefficients toward zero by a factor of $\frac{1}{1+\lambda}$. This introduces bias but reduces variance.

Why this matters: In the orthonormal case, Ridge is simply scaling down the OLS solution. For general X , the shrinkage is more complex—coefficients corresponding to directions with small eigenvalues are shrunk more than those with large eigenvalues.

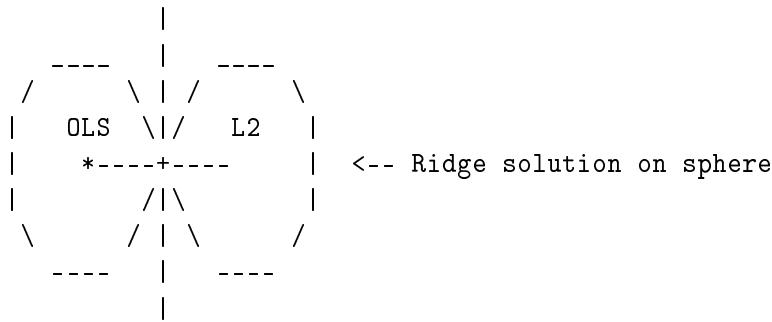
31.3.2 Geometric Interpretation of Ridge

Ridge regression can equivalently be written as a **constrained optimisation**:

$$\min_{\beta} \|y - X\beta\|_2^2 \quad \text{subject to} \quad \|\beta\|_2^2 \leq t$$

for some t that depends on λ (larger λ corresponds to smaller t).

Geometric picture: The OLS loss function defines elliptical contours centred at $\hat{\beta}_{\text{OLS}}$. The constraint $\|\beta\|_2^2 \leq t$ is a sphere centred at the origin. The ridge solution is where the smallest loss contour touches the sphere.



Since the sphere is smooth everywhere, the ridge solution is typically in the interior of any coordinate direction—all coefficients are shrunk but none are exactly zero.

31.4 Lasso Regression (L1 Regularisation)

Lasso Regression

$$\mathcal{L}_{\text{lasso}}(\beta; \lambda) = \frac{1}{n} \|y - X\beta\|_2^2 + \lambda \|\beta\|_1$$

where $\|\beta\|_1 = \sum_j |\beta_j|$ penalises the **sum of absolute values** of coefficients.

No closed-form solution—requires iterative optimisation (e.g., coordinate descent).

Why no closed-form? The L1 penalty $|\beta_j|$ is not differentiable at $\beta_j = 0$. This non-smoothness is precisely what enables sparsity (coefficients can be exactly zero), but it means we can't simply set the gradient to zero and solve.

31.4.1 Lasso as Soft Thresholding

For orthonormal X :

$$\hat{\beta}_{\text{lasso},j} = \text{sign}(\hat{\beta}_{\text{OLS},j}) \cdot (|\hat{\beta}_{\text{OLS},j}| - \lambda)_+$$

where $(z)_+ = \max(0, z)$ denotes the positive part.

Understanding the Lasso Formula

- $\text{sign}(\hat{\beta}_j^{\text{OLS}})$: Preserves the direction (positive or negative) of the coefficient
- $|\hat{\beta}_j^{\text{OLS}}| - \lambda$: Subtracts a constant λ from the absolute value
- $(\cdot)_+$: Sets result to zero if negative

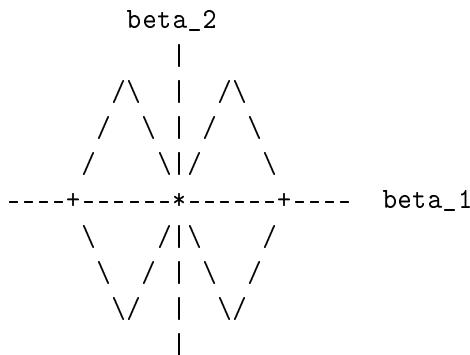
This is a **thresholding function**: take the magnitude of the OLS estimate, subtract λ , and take the positive part. Small coefficients (those with $|\hat{\beta}_j^{\text{OLS}}| < \lambda$) are set exactly to zero.

Visualising the difference:

- **Ridge**: $\hat{\beta}_{\text{ridge}} = \frac{\hat{\beta}_{\text{OLS}}}{1+\lambda}$ — multiply by a shrinkage factor (never exactly zero)
- **Lasso**: $\hat{\beta}_{\text{lasso}} = \text{sign}(\hat{\beta}_{\text{OLS}}) \cdot (|\hat{\beta}_{\text{OLS}}| - \lambda)_+$ — subtract and threshold (can be exactly zero)

31.4.2 Why Lasso Produces Sparsity: Geometric Intuition

The Lasso constraint region $\|\beta\|_1 \leq t$ is a **diamond** (in 2D) or cross-polytope (in higher dimensions). Unlike the smooth L2 ball, this constraint region has **corners** at the coordinate axes.



When the elliptical loss contours meet the diamond constraint, they are more likely to touch at a **corner** than at a smooth edge. Corners correspond to sparse solutions—coefficients on some axes are exactly zero.

Why Corners Mean Sparsity

At a corner of the L1 ball, one or more coordinates are exactly zero. The probability that an ellipse first touches a polytope at a corner (rather than a face) is positive—and in high dimensions, this probability increases. This geometric fact explains why Lasso automatically performs feature selection.

Formal intuition: The ellipse (loss contours) is smooth; the diamond (constraint) has corners. For the ellipse to touch the diamond at a non-corner point, it must be tangent to a flat face. But generically, the ellipse will hit a corner first because corners “stick out.”

Ridge vs Lasso: A Comparison

	Ridge (L2)	Lasso (L1)
Penalty	$\ \beta\ _2^2 = \sum \beta_j^2$	$\ \beta\ _1 = \sum \beta_j $
Effect	Shrinks all coefficients	Sets some coefficients to exactly 0
Sparsity	No	Yes (automatic feature selection)
Solution	Closed-form	Iterative
Geometry	Circular constraint	Diamond constraint
Orthonormal X	$\frac{\hat{\beta}_{OLS}}{1+\lambda}$	Soft thresholding

Ridge gives small $\beta^\top \beta$; **Lasso** makes β **sparse** (drives coefficients to zero).

31.5 Elastic Net

Elastic Net Regression

Combines L1 and L2 penalties:

$$\mathcal{L}_{\text{elastic}}(\beta; \lambda, \alpha) = \frac{1}{n} \|y - X\beta\|_2^2 + \lambda [\alpha \|\beta\|_1 + (1 - \alpha) \|\beta\|_2^2]$$

where $\alpha \in [0, 1]$ controls the mix between L1 and L2:

- $\alpha = 1$: Pure Lasso
- $\alpha = 0$: Pure Ridge
- $\alpha \in (0, 1)$: Elastic Net (hybrid)

Alternative parameterisation (sometimes used):

$$\mathcal{L}_{\text{elastic}}(\beta; \lambda_1, \lambda_2) = \frac{1}{n} \|y - X\beta\|_2^2 + \lambda_1 \|\beta\|_1 + \lambda_2 \|\beta\|_2^2$$

Why combine penalties? Lasso has two limitations that Ridge addresses:

1. **Correlated features:** When features are highly correlated, Lasso tends to select one arbitrarily and zero out the others. The L2 penalty encourages the coefficients of correlated

features to be similar.

2. **$n < p$ limitation:** Lasso can select at most n features (when $n < p$). Adding L2 removes this constraint.
3. **Stability:** Small changes in data can cause Lasso to select different features from a correlated group. The L2 component stabilises the selection.

When to Use Each Method

- **Ridge:** When you believe all features contribute somewhat; prediction is the goal; features are correlated
- **Lasso:** When you want interpretable sparse models; feature selection is important; features are relatively independent
- **Elastic Net:** When you want sparsity but features are correlated; $p \gg n$; stability matters

32 Multiple Perspectives on Regularisation

Section Summary

Regularisation isn't just a trick—it has deep justifications from multiple perspectives: **necessity** (making ill-posed problems solvable), **bias-variance** (trading off estimation error), **Bayesian** (encoding prior beliefs), **geometric** (constraining the feasible region), and **measurement error** (accounting for noise in features).

Regularisation can be understood from several complementary viewpoints, each providing different intuition.

32.1 Perspective 1: Necessity (Invertibility)

When OLS Fails

OLS requires $(X^\top X)^{-1}$ to exist. This fails when:

- $n < p$ (more features than observations)
- Columns of X are linearly dependent (multicollinearity)
- Near-collinearity (numerically unstable)

The requirements for invertibility:

- Non-zero determinant
- Full rank: each column of X is linearly independent
- Equivalent to an eigenvalue condition: all eigenvalues are positive

Ridge **guarantees invertibility**: $(X^\top X + \lambda I)$ is always positive definite for $\lambda > 0$.

Diagonal Dominance

The key insight is **diagonal dominance**: if $\sum_{j \neq i} |A_{ij}| < |A_{ii}|$ for all i , then A is invertible.

Adding λI makes $(X^\top X)_{ii}$ larger—the “ridge” dominates the matrix, eventually making it invertible.

$$\hat{\beta}_{\text{ridge}} = (X^\top X + \lambda I_p)^{-1} X^\top y$$

Here the scaling factor λ increases the diagonal terms, guaranteeing invertibility.

Eigenvalue perspective: If $X^\top X$ has eigenvalues $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_p \geq 0$, then $X^\top X + \lambda I$ has eigenvalues $\lambda_1 + \lambda \geq \lambda_2 + \lambda \geq \dots \geq \lambda_p + \lambda > 0$. All eigenvalues are strictly positive, guaranteeing invertibility.

32.2 Perspective 2: Bias-Variance Tradeoff

NB!

[Core Intuition] Regularisation **introduces bias** in order to **reduce variance**.

Regularisation Trades Bias for Variance

- **OLS**: Unbiased but high variance (especially with many features)
- **Ridge**: Biased (shrinks toward zero) but lower variance

When variance dominates (high-dimensional settings), this tradeoff improves MSE:

$$\text{MSE} = \text{Bias}^2 + \text{Variance}$$

A small increase in bias can yield a large decrease in variance.

Why variance can dominate: In high dimensions with limited data, OLS estimates are highly sensitive to the particular sample drawn. Different samples give wildly different coefficient estimates. By shrinking toward zero, Ridge stabilises the estimates—they become more consistent across samples (lower variance) at the cost of being systematically too small (bias).

Why Social Scientists Often Avoid Regularisation

In social science and econometrics, unbiased estimation is often prioritised:

1. **Causal interpretation**: Biased estimates can lead to incorrect causal conclusions
2. **Interpretability**: Shrinkage changes the meaning of coefficients

The philosophy is: first find an unbiased estimator, then work to reduce its variance. Ridge introduces bias deliberately (scaling down)—useful for prediction but potentially problematic for inference.

This highlights a fundamental difference between **prediction** (where bias-variance tradeoff matters) and **inference** (where unbiasedness may be paramount).

32.3 Perspective 3: Bayesian Interpretation (MAP)

NB!

[Key Connection] The Bayesian MAP estimator *is* ridge regression. There is a one-to-one correspondence between regularisation and prior distributions.

Regularisation as Prior

Ridge regression is equivalent to MAP estimation with a Gaussian prior:

$$\beta \sim \mathcal{N}(0, \tau^2 I)$$

The regularisation parameter relates to the prior: $\lambda = \sigma^2 / \tau^2$

- $\tau \rightarrow \infty$ (weak prior, large variance): Ridge \rightarrow OLS (indifferent to prior)
- $\tau \rightarrow 0$ (strong prior, small variance): $\hat{\beta} \rightarrow 0$ (ignores data, assumes $\beta = 0$)

Similarly, Lasso corresponds to a **Laplace prior** (double exponential):

$$p(\beta_j) \propto \exp(-|\beta_j|/b)$$

The Laplace distribution has heavier tails than Gaussian but concentrates more mass at zero, explaining why Lasso produces sparse solutions.

Why the equivalence holds: Bayes' theorem gives:

$$p(\beta|y, X) \propto p(y|X, \beta) \cdot p(\beta)$$

Taking the log:

$$\log p(\beta|y, X) = \log p(y|X, \beta) + \log p(\beta) + \text{const}$$

With Gaussian likelihood and Gaussian prior:

$$\begin{aligned} \log p(y|X, \beta) &\propto -\frac{1}{2\sigma^2} \|y - X\beta\|^2 \\ \log p(\beta) &\propto -\frac{1}{2\tau^2} \|\beta\|^2 \end{aligned}$$

Maximising the posterior (MAP) is equivalent to minimising:

$$\frac{1}{\sigma^2} \|y - X\beta\|^2 + \frac{1}{\tau^2} \|\beta\|^2$$

which is exactly the Ridge objective with $\lambda = \sigma^2 / \tau^2$.

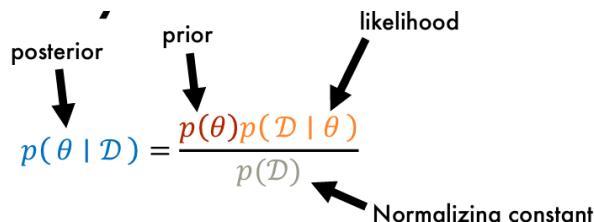


Figure 13: Bayes' theorem: the posterior $p(\theta|\mathcal{D})$ is proportional to the prior $p(\theta)$ times the likelihood $p(\mathcal{D}|\theta)$. Regularisation enters through the prior.

Frequentist-Bayesian Connection

Regularisation	Prior Distribution
L2 (Ridge)	Gaussian $\mathcal{N}(0, \tau^2)$
L1 (Lasso)	Laplace (double exponential)
Elastic Net	Mixture of Gaussian and Laplace
None (OLS)	Uniform (improper)

This relationship highlights a beautiful crossover between frequentist (regularisation) and Bayesian approaches. Choosing a specific form of regularisation implicitly makes assumptions akin to choosing a prior in Bayesian analysis.

Intuitive Understanding

If you believe the true parameters should be small (to avoid overfitting), you can express this belief by:

- **Frequentist:** Imposing a penalty on parameter size (regularisation)
- **Bayesian:** Choosing priors that favour smaller values

Both approaches add extra information to guide the optimisation toward certain properties. Regularisation does this via a penalty term; Bayesian inference does this through priors combined with the likelihood to form posteriors.

32.4 Perspective 4: Geometric Interpretation

The regularised objective can be written as a constrained optimisation:

$$\min_{\beta} \|y - X\beta\|_2^2 \quad \text{subject to} \quad \|\beta\|_p \leq t$$

- **Ridge:** Constraint region is a **sphere** (ℓ_2 ball)
- **Lasso:** Constraint region is a **diamond** (ℓ_1 ball)

The Lasso's corners at the axes explain why it produces exact zeros: the elliptical contours of the loss function are more likely to first touch the constraint at a corner, corresponding to a coordinate being exactly zero.

Lagrangian duality: The penalised form $\min \|y - X\beta\|^2 + \lambda \|\beta\|^2$ and the constrained form $\min \|y - X\beta\|^2 \text{ s.t. } \|\beta\|^2 \leq t$ are equivalent via Lagrangian duality. Each value of λ corresponds to some value of t , and vice versa.

32.5 Perspective 5: Measurement Error

Adding Gaussian noise to features is equivalent to Ridge regression:

If we observe $\tilde{X} = X + \epsilon$ where $\epsilon_{ij} \sim \mathcal{N}(0, \sigma^2)$, then OLS on \tilde{X} yields:

$$\hat{\beta} \approx (X^\top X + n\sigma^2 I)^{-1} X^\top y$$

This simplifies to the ridge regression objective:

$$R(\theta) = \frac{1}{n} \sum_{i=1}^n (X\beta - y_i)^2 + \sigma^2 \|\beta\|_2^2$$

Intuition: Noise Breaks Spurious Correlations

Adding Gaussian noise to features effectively shrinks coefficients. Why?

If you add infinite Gaussian noise to each feature, all relationships become random—there will be no linear relationship between features and output. Less linear relationship means smaller coefficients.

Features that don't truly predict y get shrunk because noisy versions of them show no relationship. Thus, adding noise is equivalent to regularisation.

Practical implication: If you believe your features are measured with error (which is almost always true in practice), Ridge regression is implicitly accounting for this measurement error. This provides another justification for why regularisation often improves predictions on real data.

33 Model Selection and Validation

Section Summary

Hyperparameters like λ cannot be learned from training data alone—we need held-out data. **Validation sets** provide unbiased estimates of generalisation error; **cross-validation** maximises data efficiency when samples are scarce. The test set must remain untouched until final evaluation to avoid optimistic bias.

How do we choose the regularisation strength λ ? This is a **hyperparameter**—a parameter that controls the learning process rather than being learned from data.

We want to choose hyperparameter values to **minimise generalisation error**.

33.1 Validation Sets

Three-Way Split

1. **Training set** (p_{train}): Fit model for each candidate λ
2. **Validation set** ($p_{\text{validation}}$): Choose λ that minimises validation loss (model selection)
3. **Test set** (p_{test}): Estimate final generalisation error (used only once!)

This prevents “leaking” test information into model selection.

Why three sets? If we use the test set to choose λ , we’re effectively fitting to the test set—our reported “test error” will be optimistically biased. The validation set acts as a proxy for test data during model selection, preserving the test set’s integrity for final evaluation.

NB!

[Never Use the Test Set for Model Selection!] If you repeatedly evaluate on the test set and choose the best model, you’re effectively fitting to the test set and will overestimate performance on truly new data.

The test set should be touched exactly once—at the very end, to report final performance.

33.2 Cross-Validation

When data is limited, cross-validation reuses data for both training and validation.

K-Fold Cross-Validation

1. Split data into K roughly equal folds
2. For $k = 1, \dots, K$:
 - Train on all folds except k
 - Evaluate on fold k
3. Average the K validation scores

Common choices: $K = 5$ or $K = 10$

Leave-One-Out CV (LOOCV): $K = n$. Uses maximum data for training but computationally expensive.

Why cross-validation works: Each data point serves as validation data exactly once and as training data $K - 1$ times. This gives us a nearly unbiased estimate of generalisation error while using all data for both training and validation (though not simultaneously).

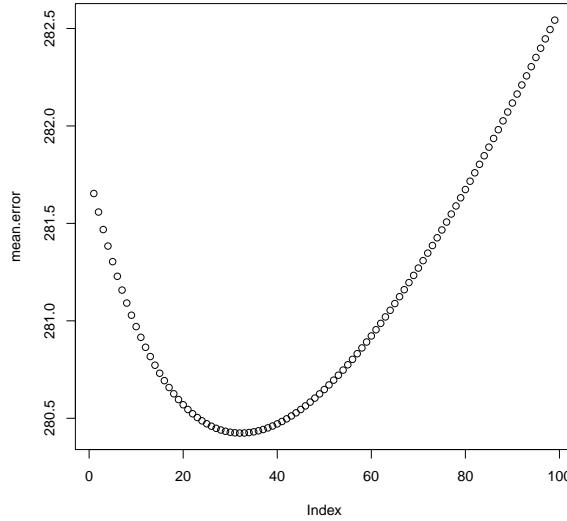


Figure 14: Cross-validation error as a function of model complexity (indexed on x-axis). The U-shaped curve shows the bias-variance tradeoff: too simple models underfit (high error on left), too complex models overfit (high error on right). The optimal complexity minimises CV error.

Choosing λ via CV

1. Define a grid of λ values (e.g., $10^{-4}, 10^{-3}, \dots, 10^2$)
2. For each λ , compute CV score
3. Select $\lambda^* = \arg \min_{\lambda} \text{CV}(\lambda)$
4. Refit on full training data with λ^*

Practical considerations:

- Use a logarithmic grid for λ (it spans many orders of magnitude)
- $K = 5$ or $K = 10$ works well in practice; LOOCV can have high variance
- Stratified folds maintain class balance in classification problems
- Some practitioners use “one standard error rule”: choose the simplest model within one standard error of the minimum CV error

34 Regularised Polynomial Regression

Section Summary

The solution to high-dimensional instability: combine expressive feature expansions with regularisation. Use high-degree polynomials for flexibility, Ridge or Lasso to control complexity, and cross-validation to tune λ . This gives smooth, stable fits without the wild oscillations of unregularised high-degree polynomials.

Combining polynomial features with regularisation gives the best of both worlds:

- **High expressivity:** Can fit complex relationships (high-dimensional model)
- **Controlled complexity:** Regularisation prevents overfitting

Recipe for Flexible Regression

1. Expand features (polynomials, interactions, etc.)
2. Apply Ridge or Lasso regularisation
3. Choose λ via cross-validation

This allows fitting smooth, complex curves without the instability of high-degree unregularised polynomials.

Why this works: The polynomial expansion provides the model with the *capacity* to fit complex functions. Regularisation then controls *how much* of this capacity is actually used, based on what the data supports. High λ effectively “turns off” the higher-order terms; low λ lets them contribute. Cross-validation finds the right balance.

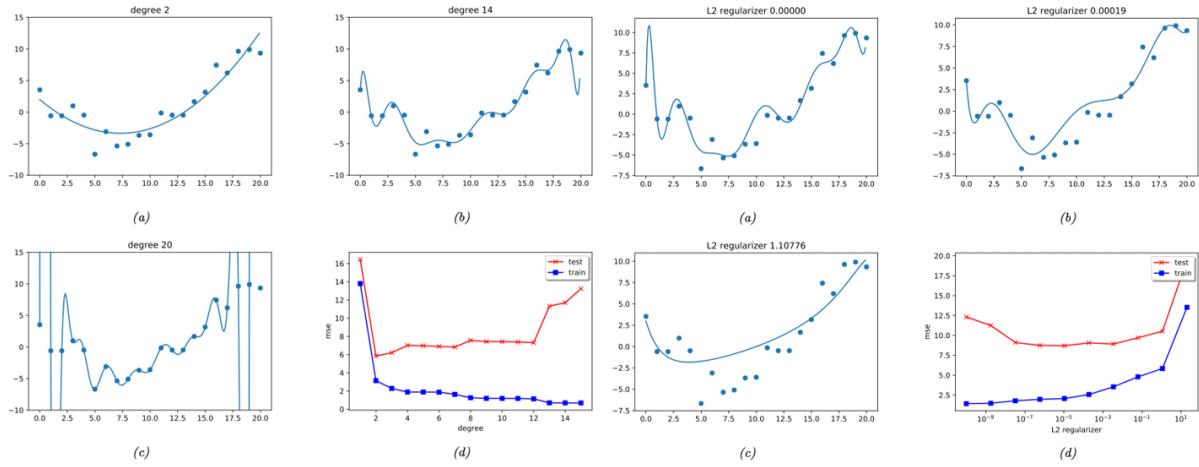


Figure 15: Regularised polynomial regression. The right panels show L2 regularisation producing smooth fits even with high-degree polynomials, avoiding the oscillatory behaviour of unregularised fits. Panel (c) is particularly notable: a smooth model that doesn't exhibit typical polynomial wiggles. Regularisation allows us to get the best of both worlds—expressivity without instability.

Connection to neural networks: This same principle—high capacity plus regularisation—underlies modern deep learning. Neural networks have enormous capacity (millions of parameters), but techniques like weight decay (L2 regularisation), dropout, and early stopping prevent overfitting. The lesson from polynomial regression generalises: flexibility is useful, but must be controlled.

35 Summary

Key Concepts from Week 3

1. **Polynomial regression:** Nonlinear relationships via feature expansion; still linear in parameters
2. **Numerical instability:** High-degree polynomials have ill-conditioned $X^\top X$; condition number measures error amplification
3. **Runge's phenomenon:** Global approximators cause edge problems; more flexibility isn't always better
4. **Curse of dimensionality:** Volume concentrates near boundaries; distances become uniform; local methods fail
5. **Population vs empirical risk:** We minimise the latter (what we can compute) to approximate the former (what we care about)
6. **Three levels of optimality:** Bayes optimal f^{**} , best-in-class f^* , empirical best \hat{f}_n
7. **Approximation error:** Limitation of hypothesis class (doesn't decrease with n)
8. **Estimation error:** Finite-sample error (decreases with n , increases with complexity)
9. **Regularisation:** Penalty on complexity to reduce overfitting; trades bias for variance
10. **Ridge (L2):** Shrinks coefficients, guarantees invertibility, closed-form solution
11. **Lasso (L1):** Sparse solutions, automatic feature selection, no closed-form
12. **Elastic Net:** Combines L1 and L2; sparsity with stability
13. **Multiple perspectives:** Regularisation as necessity, bias-variance tradeoff, Bayesian prior, geometric constraint, measurement error
14. **Bayesian view:** Ridge = Gaussian prior, Lasso = Laplace prior
15. **Cross-validation:** Choose hyperparameters without overfitting to test data

Looking Ahead

The techniques from this week—feature expansion controlled by regularisation—form the foundation for many advanced methods:

- **Kernel methods** (Week 4): Implicit feature expansion to infinite dimensions
- **Neural networks:** Learned feature representations with massive capacity
- **Sparse models:** Lasso and extensions for interpretable high-dimensional inference

The core insight persists: model flexibility must be balanced against data availability, and regularisation provides a principled way to achieve this balance.

Overview

This week addresses the fundamental question in machine learning: *how well will our model perform on data it has never seen?* The ability to generalise—to make accurate predictions on new, unseen data—is what separates useful models from those that have merely memorised their training data.

We develop both practical tools and theoretical frameworks for reasoning about generalisation:

- **Practical validation strategies:** Cross-validation techniques for hyperparameter selection and performance estimation
- **Model selection criteria:** Information-theoretic approaches (AIC, BIC) as alternatives to cross-validation
- **Theoretical bounds:** Probabilistic guarantees on generalisation error
- **Complexity measures:** VC dimension and Rademacher complexity for quantifying hypothesis class richness
- **Dimensional regimes:** How generalisation behaviour changes in low vs. high dimensional settings

The tension between fitting the training data well and generalising to new data is the central theme. A model that perfectly memorises training examples (zero training error) may perform terribly on test data—this is **overfitting**. Conversely, a model that is too simple to capture patterns in the data will have high error on both training and test data—this is **underfitting**. Our goal is to find the sweet spot.

36 Cross-Validation

Section Summary

Cross-validation estimates generalisation error by repeatedly holding out portions of training data. **K-fold CV** balances bias and variance of the estimate; **LOOCV** maximises data usage but has high variance. For linear models, the **hat matrix shortcut** computes LOOCV in $O(n)$ rather than $O(n^2)$. The **one standard error rule** implements Occam's razor by preferring simpler models within noise margins. When data has structure (groups, time ordering), use **grouped** or **time series CV** to avoid information leakage.

Cross-validation is the workhorse technique for estimating how well a model will generalise to independent data. It enables principled hyperparameter selection by providing an estimate of out-of-sample performance using only training data.

36.1 The Core Problem

We want to select hyperparameters (regularisation strength, model complexity, etc.) that will give good performance on *new* data. But we cannot use the test set for this purpose—doing so would leak information and invalidate our final evaluation. Cross-validation solves this by cleverly reusing the training data.

36.2 Why Train-Test Split Isn't Enough

A single train-test split provides a point estimate of generalisation error, but this estimate has substantial **variance**. Consider the following scenario:

- With $n = 100$ observations and an 80/20 split, you're estimating performance from just 20 data points
- A different random split could yield a substantially different estimate
- The variance of the estimate depends on the test set size: $\text{Var}(\hat{R}) \propto 1/n_{\text{test}}$

Cross-validation addresses this by averaging over multiple splits, reducing the variance of the generalisation estimate at the cost of increased computation.

What this means practically: If you report “my model achieved 85% accuracy” based on a single train-test split with a small test set, that number might be 80% or 90% with a different split. Cross-validation gives you a more stable estimate and a sense of the uncertainty.

36.3 K-Fold Cross-Validation

K-Fold Cross-Validation

Partition the training data into K roughly equal-sized **folds**. For each fold $k \in \{1, \dots, K\}$:

1. **Hold out** fold k as a validation set
2. **Train** the model on the remaining $K - 1$ folds
3. **Evaluate** the trained model on fold k

The cross-validation risk estimate averages performance across all folds:

$$R^{\text{cv}} = \frac{1}{K} \sum_{k=1}^K R(\hat{\theta}(D_{-k}), D_k) \quad (1)$$

where:

- D_k denotes the data in fold k (the held-out validation set)
- D_{-k} denotes all data *except* fold k (the training set for this iteration)
- $\hat{\theta}(D_{-k})$ are the model parameters learned from D_{-k}
- $R(\cdot, \cdot)$ is the risk (error) computed on the validation set

Unpacking the notation: The expression $\hat{\theta}(D_{-k})$ emphasises that parameters are learned from the data excluding fold k . This is crucial—the validation fold must be completely unseen during training for the estimate to be valid. The subscript notation D_{-k} (“D minus k”) is standard shorthand for “all data except fold k .”

The procedure ensures that each data point is used for validation exactly once, while contributing to training in $K - 1$ of the K iterations. This provides a relatively low-variance estimate of out-of-sample performance.

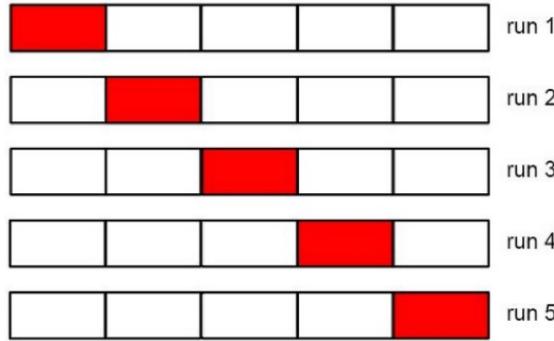


Figure 16: 5-fold cross-validation: the data is split into 5 folds, and each fold serves as the validation set exactly once while the remaining 4 folds form the training set. After all 5 iterations, every observation has been used for both training (4 times) and validation (1 time).

After cross-validation: Once you have selected hyperparameters using CV, **refit the model on all available training data** using those hyperparameters. The CV estimate tells us the expected performance; the final model should use all available information for maximum predictive power.

36.3.1 Bias-Variance Tradeoff in K-Fold CV

The choice of K involves a bias-variance tradeoff—but importantly, this is about the *CV estimate itself*, not the model’s predictions. Understanding this distinction is crucial.

Bias-Variance of the CV Estimate

Bias of CV estimate: Each fold trains on $(K - 1)/K$ of the data. When K is small:

- Training sets are smaller (e.g., 50% for $K = 2$)
- Models trained on less data perform worse than models trained on all data
- CV estimate is **pessimistically biased** (overestimates true test error of the final model trained on all data)

Variance of CV estimate: When K is large (approaching LOOCV):

- Training sets overlap heavily ($n - 1$ shared observations between any two folds)
- The K error estimates are highly correlated
- Averaging correlated estimates doesn’t reduce variance as effectively as averaging independent estimates
- CV estimate has **high variance**

Why does overlap cause correlation? Consider LOOCV where we leave out observation 1 vs. observation 2. The training sets differ by only 2 observations out of $n - 1$. The fitted models are nearly identical, so the errors on the left-out observations are highly correlated. When we average n highly correlated numbers, the variance reduction is much less than if we averaged n independent numbers.

Contrast with i.i.d. averages: If you average n independent random variables each with variance

σ^2 , the variance of the average is σ^2/n . But if the variables have correlation ρ , the variance of the average is approximately $\rho\sigma^2 + (1 - \rho)\sigma^2/n$. When ρ is close to 1 (as in LOOCV), the variance barely decreases with n .

Choosing K : Practical Guidance

- $K = 5$ or 10 : Standard choices that balance bias and variance. Empirically, $K = 10$ often performs well and is the most common choice in practice.
- $K = n$ (**LOOCV**): Nearly unbiased but high variance; useful when n is very small or when the LOOCV shortcut applies (linear models).
- $K = 2$: High bias; rarely used except for very large datasets where computation dominates and the bias matters less.

Rule of thumb: Use $K = 10$ unless you have a specific reason not to. For small datasets ($n < 50$), consider $K = n$ (LOOCV) to maximise training data.

36.3.2 Computational Cost

Computational Complexity of K-Fold CV

Let $T(n)$ denote the time to train a model on n observations.

K-fold CV: Train K models, each on $\frac{K-1}{K}n$ observations:

$$\text{Cost} = K \cdot T \left(\frac{K-1}{K}n \right) \approx K \cdot T(n)$$

LOOCV (naive): Train n models, each on $n - 1$ observations:

$$\text{Cost} = n \cdot T(n - 1) \approx n \cdot T(n)$$

For $K = 10$ and $n = 10,000$, LOOCV is **1000× more expensive** than 10-fold CV.

For linear models, the LOOCV shortcut (Section 36.4) eliminates this computational burden entirely, making LOOCV actually *cheaper* than K-fold CV.

36.4 LOOCV for Linear Regression

Leave-one-out cross-validation is particularly elegant for linear regression because the CV error can be computed **without actually refitting the model n times**. This is one of the remarkable properties of linear regression.

For linear regression, LOOCV can be computed **without refitting n times**. The key is the **hat matrix**.

The Hat Matrix

The **hat matrix** (also called the influence matrix or projection matrix) is:

$$H = X(X^\top X)^{-1}X^\top$$

This matrix “puts the hat on y ”: it projects the observed responses onto the fitted values:

$$\hat{y} = Hy$$

Key properties:

- H is symmetric: $H = H^\top$
- H is idempotent: $H^2 = H$ (projecting twice is the same as projecting once—once you’re on the regression plane, projecting again leaves you there)
- The diagonal elements $h_{ii} \in [0, 1]$ are called **leverage scores**
- $\sum_{i=1}^n h_{ii} = \text{tr}(H) = p$ (trace equals number of parameters)

Geometric intuition: The hat matrix projects from the n -dimensional space of possible response vectors onto the p -dimensional subspace spanned by the columns of X . This subspace contains all possible fitted values $X\beta$ for any choice of β .

The leverage h_{ii} measures how much observation i influences its own prediction. Points with high leverage have unusual x values—they are far from the centre of the predictor space and can disproportionately affect the fitted model.

Why “leverage”? Think of a see-saw: a person sitting far from the fulcrum has more leverage than one sitting close. Similarly, observations with extreme predictor values have more “leverage” to pull the regression line toward themselves.

LOOCV Shortcut: Derivation

We want to show that the leave-one-out residual can be computed from the full-data fit:

$$y_i - \hat{y}_i^{(-i)} = \frac{y_i - \hat{y}_i}{1 - h_{ii}}$$

where $\hat{y}_i^{(-i)}$ is the prediction for observation i from a model trained on all data *except* observation i .

Step 1: Express the leave-one-out prediction.

Let $\hat{\beta}^{(-i)}$ denote the OLS estimate excluding observation i . By the Sherman-Morrison formula for rank-one updates to matrix inverses:

$$(X_{-i}^\top X_{-i})^{-1} = (X^\top X - x_i x_i^\top)^{-1} = (X^\top X)^{-1} + \frac{(X^\top X)^{-1} x_i x_i^\top (X^\top X)^{-1}}{1 - h_{ii}}$$

What is Sherman-Morrison? When you remove one observation, the matrix $X^\top X$ changes by a rank-one update (subtracting $x_i x_i^\top$). Sherman-Morrison provides a closed-form expression for how the inverse changes, avoiding recomputation from scratch.

Step 2: Compute the difference in predictions.

After algebra (omitted), we obtain:

$$\hat{y}_i^{(-i)} = \hat{y}_i - h_{ii} \cdot \frac{y_i - \hat{y}_i}{1 - h_{ii}}$$

Step 3: Solve for the leave-one-out residual.

Rearranging:

$$y_i - \hat{y}_i^{(-i)} = y_i - \hat{y}_i + h_{ii} \cdot \frac{y_i - \hat{y}_i}{1 - h_{ii}} \tag{2}$$

$$= \frac{(1 - h_{ii})(y_i - \hat{y}_i) + h_{ii}(y_i - \hat{y}_i)}{1 - h_{ii}} \tag{3}$$

$$= \frac{y_i - \hat{y}_i}{1 - h_{ii}} \tag{4}$$

LOOCV Shortcut Formula

The LOOCV mean squared error for linear regression is:

$$\text{MSE}_{\text{cv}} = \frac{1}{n} \sum_{i=1}^n \left(\frac{y_i - \hat{y}_i}{1 - h_{ii}} \right)^2 = \frac{1}{n} \sum_{i=1}^n \left(\frac{e_i}{1 - h_{ii}} \right)^2$$

where:

- $y_i - \hat{y}_i = e_i$ is the ordinary residual from the full-data fit
- h_{ii} is the leverage of observation i

Interpretation: The denominator $1 - h_{ii}$ inflates residuals for high-leverage points. This makes sense: if a point has high leverage, removing it would change the fit substantially, so the leave-

one-out residual would be larger than the ordinary residual.

Why This Only Works for Linear Models

The derivation relies on:

1. **Closed-form solution:** OLS has $\hat{\beta} = (X^\top X)^{-1} X^\top y$
2. **Linear predictions:** $\hat{y} = X\hat{\beta}$ is linear in y
3. **Sherman-Morrison formula:** Allows efficient rank-one updates to the inverse

For nonlinear models (neural networks, tree ensembles), these properties don't hold, and we must actually refit for each fold.

However, **generalised cross-validation (GCV)** provides an approximation for some regularised linear models:

$$\text{GCV} = \frac{1}{n} \sum_{i=1}^n \left(\frac{y_i - \hat{y}_i}{1 - \bar{h}} \right)^2$$

where $\bar{h} = \text{tr}(H)/n$ is the average leverage. GCV replaces individual leverages with their average, simplifying computation further.

Computational Complexity Comparison

Method	Time Complexity	For $n = 10^4$, $p = 100$
Naive LOOCV	$O(n \cdot np^2)$	$\sim 10^{12}$ operations
LOOCV shortcut	$O(np^2 + n)$	$\sim 10^8$ operations
10-fold CV	$O(10 \cdot np^2)$	$\sim 10^9$ operations

The shortcut makes LOOCV **cheaper than K-fold CV** for linear models!

36.5 One Standard Error Rule

When comparing models using cross-validation, it is tempting to simply select the model with the lowest CV error. However, this ignores the uncertainty in our CV estimates.

One Standard Error Rule

Rather than selecting the model with minimum CV error:

1. **Compute CV error for each model:** Obtain \hat{R}_λ for each hyperparameter setting λ
2. **Find the minimum:** Let \hat{R}_{\min} be the lowest CV error observed
3. **Compute the standard error:**

$$\text{SE} = \frac{\sigma_{\text{cv}}}{\sqrt{K}} \quad (5)$$

where σ_{cv} is the standard deviation of the K fold-wise error estimates

4. **Select the simplest model** whose CV error satisfies:

$$\hat{R} \leq \hat{R}_{\min} + \text{SE} \quad (6)$$

What does “simplest” mean? This depends on the context. For Ridge or Lasso regression, simpler means **larger** λ (more regularisation, which shrinks coefficients toward zero). For polynomial regression, simpler means **lower** degree. For neural networks, simpler might mean fewer layers or smaller width.

Why prefer simpler models? This rule implements **Occam’s razor**: when multiple models have statistically indistinguishable performance (within one standard error), prefer the simpler one. Simpler models are:

- More interpretable
- Less prone to overfitting
- More likely to generalise to genuinely new situations (distribution shift)
- Often cheaper to deploy

By acknowledging uncertainty in our risk estimates, we recognise that a slightly more regularised (simpler) model is, within the margin of error, as good as the very lowest—and since we have uncertainty, we should favour the simpler one.

36.5.1 Worked Example: Ridge Regression

Example: Applying the One Standard Error Rule

Suppose we perform 10-fold CV for Ridge regression with regularisation parameter λ on a grid:

λ	CV Error	SE
0.001	0.852	0.045
0.01	0.823	0.042
0.1	0.801	0.038
1.0	0.795	0.036
10	0.812	0.041
100	0.891	0.052

Step 1: Minimum CV error is $\hat{R}_{\min} = 0.795$ at $\lambda = 1.0$.

Step 2: Standard error at minimum is $SE = 0.036$.

Step 3: Threshold is $0.795 + 0.036 = 0.831$.

Step 4: The simplest model (largest λ) with CV error ≤ 0.831 is $\lambda = 10$ (CV error = 0.812).

Selected model: $\lambda = 10$, not $\lambda = 1$.

Commentary: The model with $\lambda = 10$ has higher CV error than $\lambda = 1$, but the difference (0.017) is less than half a standard error. Given this uncertainty, we prefer the more regularised model. In practice, this often leads to better generalisation, especially when the test distribution differs slightly from the training distribution.

36.5.2 When to Use the One Standard Error Rule

When to Apply the 1SE Rule

Use when:

- Interpretability matters (simpler models are easier to understand)
- You expect distribution shift (simpler models are more robust)
- Computational cost of the model matters at deployment
- You're concerned about overfitting to the validation set itself

Don't use when:

- Maximum predictive accuracy is paramount (e.g., Kaggle competitions)
- You have very large samples (SE becomes tiny, rule has no effect)
- The “simplicity” ordering isn't meaningful for your models

NB!

[Training Error is Optimistically Biased] Training error systematically **underestimates** true out-of-sample error. The model parameters θ were optimised on the training data, making the model potentially too tailored to idiosyncrasies of the training set.

If we select hyperparameters by minimising training error:

$$\hat{\lambda} = \arg \min_{\lambda} \min_{\theta} R_{\lambda}(\theta, D) \quad (7)$$

$$= \arg \min_{\lambda} \left[\min_{\theta} R(\theta, D) + \lambda C(\theta) \right] \quad (8)$$

$$= 0 \quad (9)$$

Training error **always prefers no regularisation** ($\lambda = 0$). This is precisely why we cannot use training error for model selection—it leads us to select overly complex models.

The one standard error rule embodies a **regularisation philosophy**: when two models perform similarly, prefer the simpler one. This is the same principle underlying Ridge and Lasso penalties—we’re willing to accept slightly worse fit for reduced complexity.

36.6 Grouped Cross-Validation

When observations are not i.i.d., naive CV causes **information leakage**—the model “sees” information from the test fold during training, leading to overoptimistic performance estimates.

Violations of Independence

Consider a linear model $y_i = X_i\beta + \epsilon_i$ where the errors are correlated within groups:

$$\epsilon \sim \mathcal{N}(0, \Sigma) \quad \text{with} \quad \Sigma_{ij} \neq 0 \text{ if } c(i) = c(j) \quad (10)$$

Here $c(i)$ denotes the group (e.g., country, patient, experimental unit) to which observation i belongs. Observations from the same group are correlated; observations from different groups are independent.

Problem: If we randomly split observations across folds, related observations may end up in both training and validation sets. The model can exploit this correlation to make artificially good predictions on the validation set—predictions that would not generalise to truly new groups.

Concrete example: Suppose you’re building a model to predict patient outcomes, and you have multiple observations per patient (e.g., multiple visits). If you randomly split observations, the model might learn patient-specific patterns during training and use them to predict that same patient’s held-out visits. But at deployment, you’ll encounter *new* patients, not new visits from known patients. Your CV estimate will be overoptimistic.

- **Examples:**

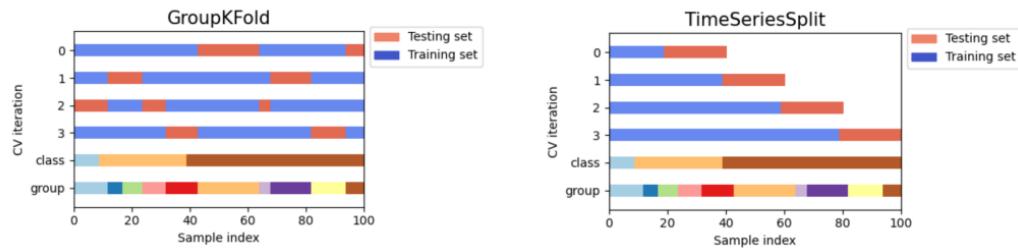


Figure 17: **Left:** Group K-fold keeps all observations from a given group together in the same fold across all CV iterations. **Right:** Time series split respects temporal ordering, always using past data to predict future data.

36.6.1 When Groups Matter

Common Grouping Scenarios

Longitudinal/Panel data: Multiple observations per subject over time.

- Medical studies: Multiple visits per patient
- User behaviour: Multiple sessions per user
- Must keep all of a subject's data in the same fold

Geographic/spatial data: Nearby observations are correlated.

- Environmental sensors in the same region
- House prices in the same neighbourhood
- Group by region, city, or spatial cluster

Hierarchical data: Natural nesting structure.

- Students within schools
- Employees within companies
- Group by the higher-level unit

Time series: Past predicts future, not vice versa.

- Financial forecasting
- Demand prediction
- Training set must precede test set temporally

36.6.2 Stratified Cross-Validation

For classification with **imbalanced classes**, random splits may put all rare-class examples in one fold, leading to folds with very different class distributions. This can cause high variance in CV estimates and poor hyperparameter selection.

Stratified K-Fold

1. Separate data by class
2. Within each class, randomly assign observations to K folds
3. Combine across classes to form final folds

Each fold has $\approx n_c/K$ examples from class c , preserving the class balance.

When to use: Binary classification with rare positives (e.g., fraud detection, disease diagnosis), or any multi-class problem where some classes are much rarer than others.

36.6.3 Nested Cross-Validation

When using CV for both hyperparameter tuning *and* performance estimation, we risk overfitting to the validation set. The hyperparameters are selected to perform well on the CV folds, so the CV estimate is optimistic about how well those hyperparameters will work on truly new data. **Nested CV** addresses this by using separate CV loops for selection and evaluation.

Nested Cross-Validation

Outer loop: Estimates generalisation error of the *entire model selection procedure*.

Inner loop: Selects hyperparameters for each outer fold.

1. Split data into K_{outer} folds
2. For each outer fold k :
 - (a) Hold out fold k as test
 - (b) On remaining data, run K_{inner} -fold CV to select best λ
 - (c) Train final model with selected λ on all non-test data
 - (d) Evaluate on test fold k
3. Average the K_{outer} test errors

This gives an unbiased estimate of expected performance when the model selection procedure is applied to new data.

Key insight: The outer loop estimates how well your *entire pipeline*—including hyperparameter selection—will perform. The hyperparameters selected in each outer fold may be different, and that's fine. We're evaluating the procedure, not a specific model.

Nested CV: Practical Considerations

- **Computational cost:** $K_{\text{outer}} \times K_{\text{inner}} \times |\text{hyperparameter grid}|$ model fits
- **Common choice:** $K_{\text{outer}} = 5, K_{\text{inner}} = 5$ ($25 \times$ cost of simple CV)
- **When to use:** When you need an unbiased performance estimate *and* hyperparameter tuning
- **Final model:** After nested CV, refit on all data using simple CV to select λ (the nested CV was just for estimation)

CV Strategies for Structured Data

Group K-Fold: All observations from a group stay together in the same fold. Use when data has natural clusters (patients, geographic regions, experimental units) and you want to predict for *new groups* not seen during training.

Time Series Split: Training set grows; test set is always in the future. Never use future data to predict the past.

Stratified K-Fold: Preserve class proportions in each fold. Use for imbalanced classification.

General principle: The CV split should mirror how the model will be deployed. If you'll predict for new groups, use group CV. If you'll predict future time points, use time series CV.

General Cross-Validation Principles

- **Information bleed:** CV should be designed so that information from one observation does not “bleed” into another—training data should not contain information that gives away answers for validation data.
- **Same fold requirement:** Observations that are related or grouped by inherent connection (same subject, same country, sequential in time) should be placed within the same fold.
- **Respect the sampling process:** When designing folds, consider the structure or dependencies in how the data was generated.

37 Model Selection Criteria

Section Summary

Information criteria (AIC, BIC) provide alternatives to cross-validation for model selection. They estimate out-of-sample prediction error by adding a complexity penalty to in-sample fit. AIC targets prediction accuracy; BIC targets identifying the true model. Both are fast to compute but rely on assumptions (likelihood framework, asymptotic approximations) that CV doesn't require.

Cross-validation is computationally expensive: we must fit the model K times (or n times for LOOCV without the shortcut). Information criteria offer closed-form alternatives that require only a single model fit.

37.1 Akaike Information Criterion (AIC)

AIC: Definition and Derivation

The **Akaike Information Criterion** estimates the expected out-of-sample deviance:

$$\text{AIC} = -2 \log L(\hat{\theta}) + 2k$$

where:

- $L(\hat{\theta})$ is the maximised likelihood
- k is the number of estimated parameters
- The factor of 2 is conventional (relates to deviance scale)

Intuition: The in-sample log-likelihood is optimistically biased. On average, the bias is approximately k (one for each fitted parameter). AIC corrects for this bias.

For linear regression with Gaussian errors:

$$\text{AIC} = n \log(\hat{\sigma}^2) + 2p$$

where $\hat{\sigma}^2 = \frac{1}{n} \sum_i (y_i - \hat{y}_i)^2$ is the residual variance and p is the number of parameters.

Unpacking the formula: The term $-2 \log L(\hat{\theta})$ measures how well the model fits the data (lower is better). The term $2k$ penalises complexity. We're trading off fit against simplicity.

Why k as the bias correction? Each parameter we estimate is “tuned” to the training data, giving us about 1 unit of optimistic bias per parameter. This is an asymptotic result that holds under certain regularity conditions.

Using AIC

- **Lower is better:** Select the model with minimum AIC
- **Relative comparisons only:** AIC values are only meaningful when comparing models on the *same data*
- **ΔAIC interpretation:** Differences in AIC can be interpreted as log-likelihood ratios; $\Delta\text{AIC} > 10$ is strong evidence against the worse model; $\Delta\text{AIC} < 2$ suggests models are roughly equivalent

37.2 Bayesian Information Criterion (BIC)

BIC: Definition

The **Bayesian Information Criterion** (also called Schwarz criterion) is:

$$\text{BIC} = -2 \log L(\hat{\theta}) + k \log n$$

The penalty term $k \log n$ grows with sample size, making BIC more conservative than AIC for large n .

Bayesian interpretation: BIC approximates the log marginal likelihood (model evidence) under certain priors. Minimising BIC approximately maximises the posterior probability of the model.

Comparing penalties: For AIC, the penalty is $2k$. For BIC, it's $k \log n$. When $n > 8$ (i.e., $\log n > 2$), BIC penalises complexity more heavily than AIC. For typical sample sizes (n in thousands), BIC is *much* more conservative.

37.3 AIC vs BIC: Different Goals

Philosophical Difference

AIC is derived from minimising Kullback-Leibler divergence to the true data-generating process. It targets **prediction accuracy**.

BIC is derived from approximating the posterior probability of a model. It targets **model identification**—finding the true model if it's in the candidate set.

Key consequences:

- AIC tends to select **larger models** (more willing to include marginally useful predictors)
- BIC tends to select **smaller models** (requires stronger evidence to include predictors)
- As $n \rightarrow \infty$: BIC is **consistent** (selects true model with probability approaching 1 if it's in the set), AIC is not
- For finite n : AIC often gives better **predictions**, even if it selects the “wrong” model

When is the “wrong” model better for prediction? If the true model is complex, AIC might select a close approximation that predicts well. BIC might select a simpler model that's worse for prediction but happens to be “true” in a limiting sense. For practical prediction, we often care more about accuracy than truth.

AIC vs BIC: Summary

	AIC	BIC
Penalty	$2k$	$k \log n$
Goal	Prediction	Model identification
Consistency	No	Yes
Typical selection	Larger models	Smaller models
Use when	Prediction matters most	Parsimony matters most

37.4 Comparison with Cross-Validation

Information Criteria vs Cross-Validation

Advantages of AIC/BIC:

- **Fast:** Single model fit, closed-form formula
- **Deterministic:** No random variation from fold assignment
- **Theoretically grounded:** Clear optimality properties under assumptions

Advantages of CV:

- **Model-agnostic:** Works for any model, any loss function
- **No distributional assumptions:** Doesn't require likelihood specification
- **Handles complex model selection:** Works for neural networks, ensembles, etc.
- **Direct estimate:** Estimates exactly what you care about (test error)

When they differ: AIC/BIC assume the model is correctly specified and errors are well-behaved. When these assumptions fail, CV is more robust. For simple models (linear regression, GLMs) on well-behaved data, they often agree.

NB!

[Limitations of AIC and BIC] AIC and BIC are **not** appropriate for:

- Comparing models with different response variables
- Comparing models fitted to different data subsets
- Models without a proper likelihood (e.g., some machine learning methods)

They are only meaningful for comparing models fitted to the **same observations** with the **same response**.

38 Frequentist vs Bayesian Risk

Section Summary

Frequentist and Bayesian frameworks conceptualise risk differently. Frequentists treat parameters as fixed unknowns and average over repeated sampling; Bayesians treat parameters as random and average over prior uncertainty. Neither framework is “correct”—they answer different questions. The choice depends on whether you want guarantees over repeated experiments (frequentist) or coherent probability statements given your data (Bayesian).

The definitions of risk we have used so far are **frequentist**. It is worth contrasting this with the Bayesian perspective to understand what each framework assumes and what questions each answers.

Frequentist Risk

$$R(\theta, f) = \mathbb{E}_{p(x|\theta)}[\ell(\theta, f(x))]$$

Key features:

- θ is **fixed but unknown**—there is a true value, we just don’t know it
- Expectation is over the **data** x drawn from $p(x|\theta)$
- Risk measures average loss across repeated sampling from the same data-generating process
- We imagine running the experiment many times and averaging performance

Unpacking the notation: The expression $\mathbb{E}_{p(x|\theta)}$ means “average over all datasets that could be generated from the true distribution $p(x|\theta)$.” We’re asking: if we repeatedly drew data from the true distribution and applied our decision function f , what would our average loss be?

Bayes Risk

$$R_{\pi_0}(f) = \mathbb{E}_{\pi_0(\theta)}[R(\theta, f)] = \int \pi_0(\theta) p(x|\theta) \ell(\theta, f(x)) d\theta dx$$

Key features:

- θ is treated as a **random variable** with prior distribution $\pi_0(\theta)$
- Expectation is over **both** θ and x
- Averages performance across all possible parameter values, weighted by their prior probability
- Incorporates prior beliefs about likely parameter values

Unpacking the Bayes risk integral: The expression $\int \pi_0(\theta) p(x|\theta) \ell(\theta, f(x)) d\theta dx$ computes a weighted average:

1. For each potential value of θ , determine the likelihood of observing data x

2. Compute the loss for decision function f given that θ and x
3. Weight this loss by the joint probability of θ and x (coming from prior \times likelihood)
4. Integrate (sum) across all possible θ and all possible x

Frequentist vs Bayesian: Key Differences

	Frequentist	Bayesian
Parameters	Fixed, unknown	Random variable
Data	Random variable	Observed (then fixed)
Prior information	Not formally used	Encoded in $\pi_0(\theta)$
Uncertainty about	Data we might see	Parameter values

Which is “right”? Neither—they answer different questions:

- **Frequentist**: “How will my procedure perform across many experiments?”
- **Bayesian**: “Given my prior beliefs and this data, what should I believe about the parameters?”

In practice, many methods that seem frequentist (like Ridge regression) have Bayesian interpretations, and vice versa. The distinction matters most for interpretation and when making decisions under uncertainty.

39 Generalisation Bounds

Section Summary

Generalisation bounds provide theoretical guarantees on how well empirical risk approximates true risk. The key tools are **Hoeffding’s inequality** (concentration of sample means) and the **union bound** (probability of any bad event). Together, they show that generalisation error scales as $O(\sqrt{\log |\mathcal{H}|}/n)$ for finite hypothesis classes. For infinite classes, we need richer complexity measures like **VC dimension** and **Rademacher complexity**.

We now turn to *theoretical* guarantees about generalisation. The goal is to bound, with high probability, how well a learned model will perform on unseen data.

39.1 Error Decomposition Recap

Error Decomposition

Recall the fundamental decomposition of excess risk:

$$\underbrace{\mathbb{E}[\mathcal{R}(f_n^*)] - \mathcal{R}(f^{**})}_{\text{Total excess risk}} = \underbrace{\mathcal{R}(f^*) - \mathcal{R}(f^{**})}_{\text{Approximation error}} + \underbrace{\mathbb{E}[\mathcal{R}(f_n^*)] - \mathcal{R}(f^*)}_{\text{Estimation error}}$$

where:

- $f^{**} = \arg \min_f \mathcal{R}(f)$: The **Bayes optimal** predictor—best possible function over all measurable functions
- $f^* = \arg \min_{f \in \mathcal{H}} \mathcal{R}(f)$: Best function within our hypothesis class \mathcal{H}
- $f_n^* = \arg \min_{f \in \mathcal{H}} \hat{\mathcal{R}}_n(f)$: Empirical risk minimiser from our data

Unpacking the terms:

- **Approximation error** ($\mathcal{R}(f^*) - \mathcal{R}(f^{**})$): The cost of restricting to hypothesis class \mathcal{H} . If the true relationship is a polynomial of degree 5 but we only consider linear functions, the best linear function will still have some irreducible error.
- **Estimation error** ($\mathbb{E}[\mathcal{R}(f_n^*)] - \mathcal{R}(f^*)$): The cost of learning from finite data. Even if \mathcal{H} contains the true function, we might not find it with limited samples.

Concrete Example: Truncating Polynomials

Suppose the true model is $y = \beta_0 + \beta_1 x + \beta_2 x^2 + \epsilon$ with $x \sim \text{Unif}(0, 1)$.

- **Best possible function:** $f^{**} = \beta_0 + \beta_1 x + \beta_2 x^2$
- **Best in our hypothesis class** (linear functions only): $f^* = \beta_0^* + \beta_1^* x$
- **Empirical estimate:** $f_n^* = \hat{\beta}_0 + \hat{\beta}_1 x$

The **approximation error** comes from neglecting the x^2 term—our hypothesis class cannot capture the true curvature. The **estimation error** comes from estimating β_0^*, β_1^* from finite, noisy data.

Generalisation bounds focus primarily on the **estimation error**: given that we are restricted to \mathcal{H} , how close can we get to the best function in \mathcal{H} ?

NB!

[Bounds Are Not Replacements for Empirical Validation] Generalisation bounds are typically too loose to be directly useful for predicting actual model performance. They help us:

- **Reason about model complexity:** Understand tradeoffs between hypothesis class richness and sample size
- **Understand scaling:** How does performance vary with n ? With model complexity?
- **Identify assumptions:** What conditions are needed for good generalisation?
- **Worst-case guarantees:** Even under unfavourable conditions, error will not exceed a threshold

For actual model selection, use cross-validation. Bounds provide conceptual understanding, not practical estimates.

39.2 Building Blocks: Concentration Inequalities

To derive generalisation bounds, we need tools that quantify how sample averages concentrate around their expectations.

Hoeffding's Inequality

For $X_1, \dots, X_n \stackrel{\text{iid}}{\sim} \text{Bernoulli}(\mu)$, the sample mean $\bar{X} = \frac{1}{n} \sum_{i=1}^n X_i$ satisfies:

$$P(|\bar{X} - \mu| > \epsilon) \leq 2 \exp(-2n\epsilon^2)$$

Interpretation: The probability that the sample mean deviates from the true mean by more than ϵ decreases **exponentially** in n (sample size) and ϵ^2 (squared tolerance).

Key implications of Hoeffding's inequality:

- **The chance is small:** As n increases, the probability of large deviation becomes exponentially smaller
- **Law of large numbers:** With more data, sample means concentrate tightly around true means
- **Quantitative control:** We can bound the probability of any specific deviation level

Example: With $n = 100$ samples, the probability that $|\bar{X} - \mu| > 0.1$ is at most $2e^{-2} \approx 0.27$. With $n = 1000$, this drops to $2e^{-20} \approx 4 \times 10^{-9}$ —essentially impossible.

TL;DR: Hoeffding's Inequality

As sample size n increases, sample estimates converge to their true population values, and we can precisely quantify how unlikely large deviations are. The convergence is **exponentially fast** in n .

Union Bound (Boole's Inequality)

For any events $\mathcal{E}_1, \dots, \mathcal{E}_d$:

$$P\left(\bigcup_{i=1}^d \mathcal{E}_i\right) \leq \sum_{i=1}^d P(\mathcal{E}_i)$$

The probability that *at least one* event occurs is at most the sum of the individual probabilities.

Why is this useful? We want to control the probability of *any* hypothesis having bad generalisation. If each hypothesis f_i has a small probability of bad generalisation, the union bound lets us control the probability that *some* hypothesis is bad.

When is the bound tight? The union bound is exact when events are mutually exclusive (no overlap). It's loose when events overlap significantly (we double-count the intersection). For generalisation bounds, the events are often highly correlated (similar hypotheses tend to fail together), so the bound is usually loose.

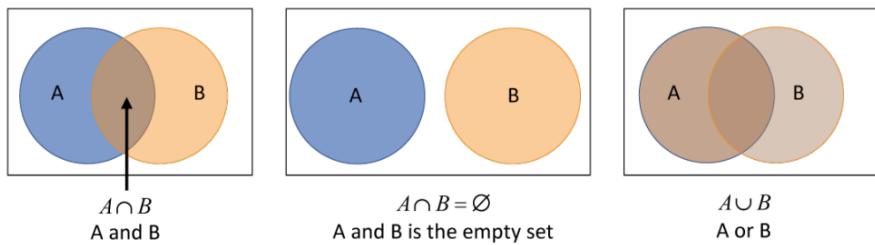


Figure 18: Union bound: $P(A \cup B) \leq P(A) + P(B)$. The bound is loose when events overlap significantly (we double-count the intersection $A \cap B$), but it avoids computing complex intersections.

39.3 First Generalisation Bound

Combining Hoeffding's inequality and the union bound yields our first generalisation guarantee.

Generalisation Bound for Finite Hypothesis Class

For binary classification with a **finite** hypothesis class \mathcal{H} :

$$P\left(\max_{f \in \mathcal{H}} |\mathcal{R}(f) - \hat{\mathcal{R}}_n(f)| > \epsilon\right) \leq 2|\mathcal{H}| \cdot \exp(-2n\epsilon^2)$$

Equivalently: with probability at least $1 - \delta$, for all $f \in \mathcal{H}$ simultaneously:

$$|\mathcal{R}(f) - \hat{\mathcal{R}}_n(f)| \leq \sqrt{\frac{\log(2|\mathcal{H}|/\delta)}{2n}}$$

Reading this bound: The statement says that with high probability, the empirical risk $\hat{\mathcal{R}}_n(f)$ is close to the true risk $\mathcal{R}(f)$ *simultaneously for all hypotheses* in \mathcal{H} . This “uniform convergence” is crucial: we need the guarantee to hold for the hypothesis we select, not just for a fixed hypothesis.

Proof Sketch

$$P \left(\max_{f \in \mathcal{H}} |\mathcal{R}(f) - \hat{\mathcal{R}}_n(f)| > \epsilon \right) = P \left(\bigcup_{f \in \mathcal{H}} \{|\mathcal{R}(f) - \hat{\mathcal{R}}_n(f)| > \epsilon\} \right) \quad (11)$$

$$\leq \sum_{f \in \mathcal{H}} P(|\mathcal{R}(f) - \hat{\mathcal{R}}_n(f)| > \epsilon) \quad (\text{Union bound}) \quad (12)$$

$$\leq \sum_{f \in \mathcal{H}} 2 \exp(-2n\epsilon^2) \quad (\text{Hoeffding for each } f) \quad (13)$$

$$= 2|\mathcal{H}| \cdot \exp(-2n\epsilon^2) \quad (\text{Finiteness of } \mathcal{H}) \quad (14)$$

Interpreting the Bound

- **$|\mathcal{H}|$ (hypothesis space size):** Larger hypothesis spaces \Rightarrow looser bounds. More models means more chances to overfit, so we need more data to get the same guarantee.
- **$\exp(-2n\epsilon^2)$ (sample size effect):** More data \Rightarrow tighter bounds. Improvement is exponential in n .
- **ϵ (tolerance):** Smaller tolerance \Rightarrow looser bounds. It is harder to guarantee small errors.

The fundamental tradeoff: Generalisation error increases with hypothesis space size but decreases with sample size. Richer model classes need more data.

39.4 Limitations of This Bound

1. **Finite hypothesis class required:** What about continuous parameters? Linear regression has infinitely many possible β values.
2. **i.i.d. assumption:** Data must be independent and identically distributed.
3. **Looseness:** Hoeffding and union bounds may not be tight; the actual generalisation error is often much better than the bound suggests.

These limitations motivate more sophisticated complexity measures like VC dimension and Rademacher complexity.

40 Measuring Hypothesis Class Complexity

Section Summary

Model complexity isn't simply "number of parameters." **VC dimension** measures the largest set of points a hypothesis class can shatter (perfectly classify under any labelling). **Rademacher complexity** measures ability to fit random labels. Both capture the true flexibility of a model class and yield meaningful generalisation bounds, even for infinite-dimensional hypothesis classes.

The generalisation bound above used $|\mathcal{H}|$ to measure complexity, but this only works for finite hypothesis classes. How do we quantify the “richness” of infinite hypothesis classes like all linear classifiers in \mathbb{R}^d ?

Approaches to Measuring Complexity

- **Parameter counting:** Number of free parameters (degrees of freedom)—simple but can be misleading
- **Smoothness measures:** Derivative-based measures (e.g., Sobolev norms), quantifying “wiggliness”
- **VC dimension:** Largest set of points that can be “shattered”—captures what the hypothesis class can represent
- **Rademacher complexity:** Ability to fit random noise—captures effective flexibility on the data

40.1 Intrinsic Dimensionality and the Manifold Hypothesis

Before discussing VC dimension, it is worth noting that the *effective* complexity of a learning problem may be much lower than it appears.

Intrinsic Dimensionality

The **intrinsic dimensionality** of a dataset is the minimum number of parameters needed to accurately describe every point—the true “complexity” of the data, as opposed to the ambient dimension it is embedded in.

Examples:

- A circle in 2D has intrinsic dimension 1 (just need angle θ)
- Earth’s surface in 3D has intrinsic dimension 2 (latitude and longitude suffice)
- Face images in 10^6 -dimensional pixel space may vary along only ~ 50 meaningful dimensions (pose, lighting, identity, expression)

Manifold Hypothesis

High-dimensional data often lies on or near a low-dimensional **manifold**—a surface that locally resembles Euclidean space of lower dimension.

Implications:

- High ambient dimensionality may mask low underlying intrinsic dimensionality
- Learning may be easier than the nominal dimension suggests
- Motivates dimensionality reduction techniques (PCA, t-SNE, autoencoders)

Example: Location predicting vote choice. Suppose we want to predict binary voting preferences from location features (latitude, longitude, height). If each coefficient β_i can be $\{-1, 0, 1\}$:

- With 3 features: $|\mathcal{H}| = 3^3 = 27$ models
- With 2 features: $|\mathcal{H}| = 3^2 = 9$ models

The bound with 2 features is tighter. But is “height” truly adding information, or is it approximately determined by latitude and longitude (i.e., redundant)? If height is redundant, dropping it reduces hypothesis space complexity *without* increasing approximation error—a pure win for generalisation.

40.2 VC Dimension

The Vapnik-Chervonenkis (VC) dimension provides a more principled measure of hypothesis class complexity that applies to infinite classes.

Shattering

A hypothesis class \mathcal{H} **shatters** a set of n points $\{x_1, \dots, x_n\}$ if, for every possible labelling $(y_1, \dots, y_n) \in \{0, 1\}^n$, there exists some $f \in \mathcal{H}$ that correctly classifies all points:

$$f(x_i) = y_i \quad \text{for all } i = 1, \dots, n$$

In other words, \mathcal{H} can achieve zero training error on these n points regardless of how they are labelled.

Intuition: Shattering measures the “expressiveness” of \mathcal{H} . If \mathcal{H} can shatter n points, it can memorise any labelling of those points. This is both a strength (flexibility) and a weakness (potential for overfitting).

VC Dimension

The **VC dimension** $\text{VC}(\mathcal{H})$ is the largest number of points that can be shattered by \mathcal{H} :

$$\text{VC}(\mathcal{H}) = \max\{n : \exists \{x_1, \dots, x_n\} \text{ that } \mathcal{H} \text{ can shatter}\}$$

If \mathcal{H} can shatter arbitrarily large sets, we say $\text{VC}(\mathcal{H}) = \infty$.

Key point: To show $\text{VC}(\mathcal{H}) \geq n$, we need to find *some* set of n points that can be shattered. To show $\text{VC}(\mathcal{H}) < n$, we need to show that *no* set of n points can be shattered.

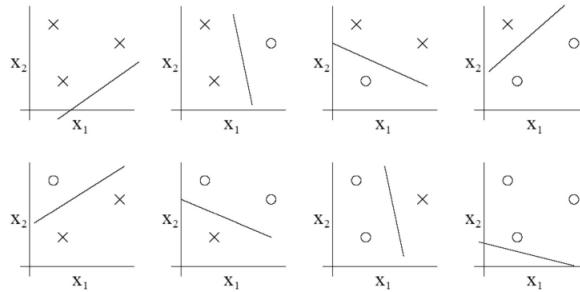


Figure 19: Linear classifiers in 2D can shatter 3 points in general position: for any labelling of the 3 points, we can find a line that separates them correctly. Hence $\text{VC}(\text{linear classifiers in } \mathbb{R}^2) \geq 3$.

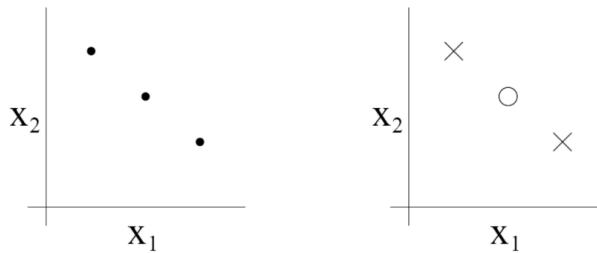


Figure 20: However, 4 points in general position *cannot* be shattered by linear classifiers in 2D. The “XOR” labelling (opposite corners have the same label) cannot be achieved by any line. Hence $\text{VC}(\text{linear classifiers in } \mathbb{R}^2) = 3 = d + 1$.

NB!

[VC Dimension \neq Parameter Count] A common misconception is that VC dimension equals the number of parameters. This is often approximately true but not always!

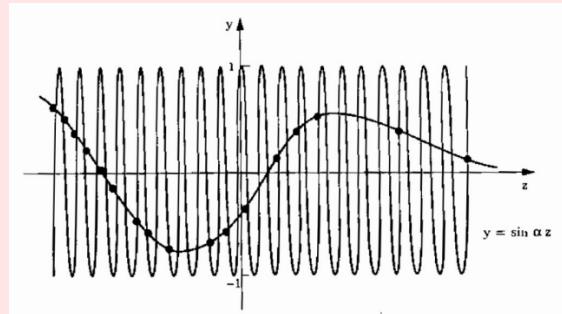


Figure 21: The function $f(x) = \text{sign}(\sin(\omega x))$ has only **one parameter** (ω) but **infinite VC dimension**. By choosing ω large enough, the function oscillates rapidly enough to shatter arbitrarily many points on the real line.

This counterexample shows that counting parameters can dramatically underestimate the flexibility of a hypothesis class. The sine function is highly “wiggly” and can fit arbitrary patterns despite having only one tunable parameter.

VC Dimension for Common Hypothesis Classes

- **Linear classifiers in \mathbb{R}^d :** $\text{VC} = d + 1$
- **Axis-aligned rectangles in \mathbb{R}^d :** $\text{VC} = 2d$
- **Neural networks:** Roughly $O(W \log W)$ where W is the number of weights (though this depends on architecture details)
- $\sin(\omega x)$: $\text{VC} = \infty$ despite having 1 parameter

40.3 The VC Bound

The VC dimension allows us to state generalisation bounds for infinite hypothesis classes, replacing $|\mathcal{H}|$ with $\text{VC}(\mathcal{H})$.

VC Generalisation Bound

With probability at least $1 - \delta$:

$$\mathcal{R}(f_n^*) - \mathcal{R}(f^*) \leq \sqrt{\frac{1}{n} \left[V \left(\log \frac{2n}{V} + 1 \right) - \log \frac{\delta}{4} \right]}$$

where $V = \text{VC}(\mathcal{H})$.

For large n , the bound scales as $O\left(\sqrt{\frac{V \log n}{n}}\right)$.

Interpretation: The estimation error decreases as $O(1/\sqrt{n})$ with sample size, but increases with VC dimension. Richer hypothesis classes (higher V) need more data to achieve the same generalisation guarantee.

Comparison to finite case: For a finite hypothesis class, we had $O(\sqrt{\log |\mathcal{H}|/n})$. For infinite classes with VC dimension V , we have $O(\sqrt{V \log n/n})$. The VC dimension replaces $\log |\mathcal{H}|$ —it's the “effective” log-size of the hypothesis class.

40.4 Rademacher Complexity

VC dimension is a worst-case measure over all possible data distributions. **Rademacher complexity** provides a data-dependent measure that can give tighter bounds for specific datasets.

Rademacher Complexity: Definition

The **empirical Rademacher complexity** of a hypothesis class \mathcal{H} with respect to data x_1, \dots, x_n measures its ability to fit random noise.

Generate random labels $\sigma_1, \dots, \sigma_n \in \{-1, +1\}$ uniformly at random (Rademacher random variables). The empirical Rademacher complexity is:

$$\hat{\mathcal{R}}_n(\mathcal{H}) = \mathbb{E}_\sigma \left[\sup_{f \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^n \sigma_i f(x_i) \right]$$

This measures how well the best function in \mathcal{H} can correlate with purely random labels.

Unpacking the definition: The inner term $\frac{1}{n} \sum_{i=1}^n \sigma_i f(x_i)$ is the empirical correlation between f 's predictions and random labels. The sup finds the hypothesis that best fits these random labels. The expectation averages over all possible random labellings.

Rademacher Complexity: Intuition

- If \mathcal{H} is very flexible, it can fit random labels well: high Rademacher complexity
- If \mathcal{H} is constrained, it cannot fit random labels: low Rademacher complexity
- Unlike VC dimension, Rademacher complexity is **data-dependent**—it can capture structure in the data distribution
- A hypothesis class might have low Rademacher complexity on “nice” data even if it has high VC dimension

Connection to overfitting: If a model class can fit random noise well (high Rademacher complex-

ity), it can also memorise spurious patterns in real data—this is exactly overfitting. Rademacher complexity directly measures this tendency.

Rademacher Generalisation Bound

With probability at least $1 - \delta$:

$$\mathcal{R}(f_n^*) \leq \hat{\mathcal{R}}_n(f_n^*) + 2\hat{\mathcal{R}}_n(\mathcal{H}) + 3\sqrt{\frac{\log(2/\delta)}{2n}}$$

The generalisation gap is controlled by the Rademacher complexity.

41 Structural Risk Minimisation

Given generalisation bounds, one might consider directly minimising them rather than using cross-validation:

$$\hat{f} = \arg \min_{f \in \mathcal{H}} [\hat{\mathcal{R}}_n(f) + \text{complexity penalty}]$$

This is **structural risk minimisation** (SRM). The idea is to balance empirical performance against hypothesis class complexity in a principled way derived from theory.

Connection to regularisation: SRM provides theoretical justification for regularisation. The complexity penalty in SRM corresponds to the regularisation term ($\lambda\|\beta\|^2$ in Ridge, $\lambda\|\beta\|_1$ in Lasso). We're adding a complexity cost to the objective, exactly as the theory recommends.

In practice: Cross-validation usually works better because:

- Theoretical bounds are often very loose (orders of magnitude)
- Cross-validation adapts to the actual data distribution
- SRM requires knowing or estimating the complexity measure, which can be hard

However, SRM provides conceptual insight: good learning algorithms should trade off fit against complexity.

42 Generalisation in Linear Regression

Section Summary

In linear regression, generalisation behaviour depends critically on the ratio p/n . In the **low-dimensional regime** ($p \ll n$), excess risk scales as $\frac{p\sigma^2}{n}$ and is dominated by variance. In the **high-dimensional regime** ($p \gg n$), bias dominates and the minimum-norm solution generalises differently. The transition at $p \approx n$ produces the **double descent** phenomenon, challenging classical intuitions about model complexity.

We now examine generalisation specifically in the context of ordinary least squares (OLS) linear regression, where we can derive precise expressions for the estimation error.

42.1 OLS Estimation Error

OLS Estimator Decomposition

The OLS estimator $\hat{\beta} = (X^\top X)^{-1} X^\top y$ can be decomposed as:

$$\hat{\beta} = \beta^* + (X^\top X)^{-1} X^\top \epsilon$$

where β^* are the true parameters and ϵ is the noise vector.

The **estimation error** $\hat{\beta} - \beta^* = (X^\top X)^{-1} X^\top \epsilon$ depends on:

- The design matrix X (specifically, how well-conditioned $X^\top X$ is)
- The noise ϵ

Key insight: The estimation error is linear in the noise. If $X^\top X$ is “close to singular” (has small eigenvalues), then $(X^\top X)^{-1}$ has large eigenvalues, amplifying the noise. This is the source of high variance in regression with correlated predictors.

OLS is the Best Linear Unbiased Estimator (BLUE) under the Gauss-Markov assumptions. The estimation error vanishes in expectation ($\mathbb{E}[\hat{\beta}] = \beta^*$), but its variance depends on the design matrix and noise level.

42.2 Singular Value Decomposition (SVD)

To analyse generalisation in different dimensional regimes, we need the singular value decomposition, which reveals the geometric structure of the design matrix.

Singular Value Decomposition

Any $n \times p$ matrix X of rank $r \leq \min(n, p)$ can be decomposed as:

$$X = U\Sigma V^\top$$

where:

- U is $n \times r$ with orthonormal columns: $U^\top U = I_r$
- Σ is $r \times r$ diagonal with singular values $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > 0$
- V is $p \times r$ with orthonormal columns: $V^\top V = I_r$

What each component represents:

- The columns of V are the **principal directions** in predictor space—the directions along which X varies most
- The singular values σ_i quantify **how much variance** exists along each direction
- The columns of U are the **corresponding directions in observation space**

$$\begin{array}{c} \boxed{\mathbf{X}} \\ (n, p) \end{array} = \begin{array}{c} \boxed{\mathbf{U}} \\ (n, r) \end{array} \begin{array}{c} \boxed{\mathbf{D}} \\ (r, r) \end{array} \begin{array}{c} \boxed{\mathbf{V}^T} \\ (r, p) \end{array}$$

Figure 22: SVD dimensions: X is $n \times p$, decomposed into U ($n \times r$), Σ ($r \times r$), and V^T ($r \times p$) where $r = \text{rank}(X)$.

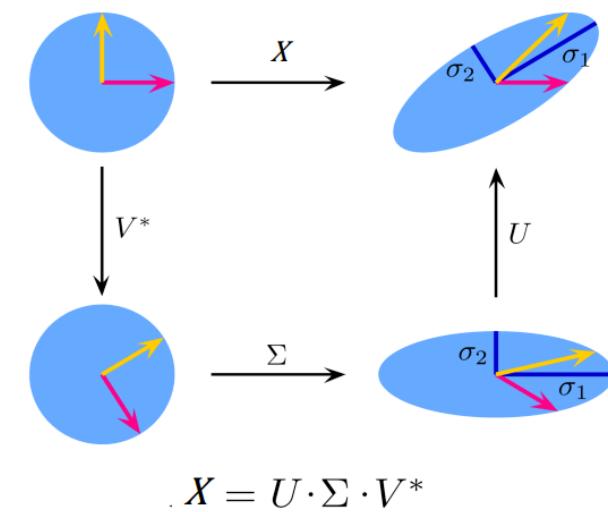


Figure 23: Geometric interpretation of SVD: any linear transformation can be decomposed as V^T (rotate input), Σ (scale along axes), U (rotate to output). SVD decomposes any linear transformation into rotation-scale-rotation.

Key SVD Properties

- $X^T X = V \Sigma^2 V^T$ (eigendecomposition of $X^T X$, with eigenvalues σ_i^2)
- $XX^T = U \Sigma^2 U^T$ (eigendecomposition of XX^T)
- OLS predictions on training data: $\hat{y} = X\hat{\beta} = UU^T y$ (projection onto column space of X)
- Predictions on new data \tilde{X} : $\tilde{X}\hat{\beta} = \tilde{X}V\Sigma^{-1}U^T y$

SVD provides a numerically stable way to compute the pseudo-inverse, even when $X^T X$ is nearly singular or exactly singular.

Why SVD matters: The SVD reveals the “directions” along which X has variance (the columns of V) and how much variance exists in each direction (the singular values σ_i). Small singular values indicate near-collinearity, which inflates estimation variance.

43 Low vs High Dimensional Regimes

The behaviour of OLS depends critically on the relationship between p (number of features) and n (number of observations).

43.1 Low-Dimensional Regime: $p \ll n$

This is the “classical statistics” regime where we have many more observations than parameters.

Expected Risk in Low Dimensions

When $p \ll n$ and standard OLS assumptions hold (with isotropic design):

$$R(\hat{\beta}) - R(\beta^*) = \frac{p}{n} \sigma^2$$

where σ^2 is the noise variance.

Unpacking this formula:

- The excess risk is the variance of the estimator (no bias for OLS)
- Each parameter contributes σ^2/n to the variance
- With p parameters, total variance is $p\sigma^2/n$

Low dimensional features

- Consider the expected error over a new test point, \tilde{x}
- $\mathbb{E}[(\tilde{x}(X^\top X)^{-1}X^\top e)^2] = \mathbb{E}[(\tilde{x}V^\top D^{-1}U^\top e)^2]$
- We can rewrite this as $\mathbb{E}\left[\left(\sum_{i=1}^p \frac{v_i^\top e u_i \tilde{x}}{d}\right)^2\right] = \sum_{i=1}^p \frac{\mathbb{E}[v_i^\top e]^2 \mathbb{E}[(u_i \tilde{x})^2]}{d_i^2}$
- $= \sum_{i=1}^p \frac{v_i^\top \mathbb{E}[ee^\top] v_i u_i^\top \mathbb{E}[\tilde{x}\tilde{x}^\top] u_i}{d_i^2} = \sigma^2 \sum_{i=1}^p \frac{u_i^\top \mathbb{E}[\tilde{x}\tilde{x}^\top] u_i}{d_i^2} = \sigma^2 \sum_{i=1}^p \frac{\text{Var}(u_i^\top \tilde{x})}{d_i^2}$
- Assume \tilde{x} is from the same distribution as X . Then,
- $\text{Var}(u_i^\top \tilde{x}) = u_i^\top \Sigma u_i = \frac{u_i^\top X X^\top u_i}{n} = \frac{u_i^\top U D^2 U^\top u_i}{n} = \frac{d_i^2}{n}$
- So $R(\hat{\beta}) - R(\beta^*) = \sigma^2 \sum_{i=1}^p \frac{1}{n} = \frac{p}{n} \sigma^2$

Figure 24: Low-dimensional regime: as n increases, the excess risk decreases smoothly as $1/n$. More data leads to rapid improvement in generalisation.

Interpretation:

- Risk scales linearly with p : more parameters \Rightarrow more estimation error
- Risk scales inversely with n : more data \Rightarrow less error
- The ratio p/n is the key quantity controlling generalisation
- Adding data provides **rapid** improvement ($\propto 1/n$)

43.2 High-Dimensional Regime: $p > n$

When $p > n$, the system is underdetermined—there are infinitely many β values that perfectly interpolate the training data (achieve zero training error). OLS as typically defined fails because $(X^\top X)^{-1}$ does not exist (the matrix is rank-deficient).

However, the **minimum-norm** (Moore-Penrose pseudo-inverse) solution can still be computed. Among all interpolating solutions, this selects the one with smallest $\|\beta\|$.

Expected Risk in High Dimensions

When $p > n$ using the minimum-norm OLS solution:

$$R(\hat{\beta}) - R(\beta^*) \approx \left(1 - \frac{n}{p}\right) \|\beta^*\|^2 + \frac{n}{p} \sigma^2$$

Unpacking this formula:

- The first term $(1 - n/p)\|\beta^*\|^2$ is essentially **bias**—the minimum-norm solution shrinks coefficients toward zero
- The second term $\frac{n}{p}\sigma^2$ is the **variance** contribution
- As $p \rightarrow \infty$ with n fixed: bias dominates, approaching $\|\beta^*\|^2$
- As n increases with p fixed: both terms decrease

High dimensional features

- **What if $p \gg n$?**
- **Standard result:**
 - Bias: $\approx \left(1 - \frac{n}{p}\right) \|\beta^*\|^2$ (**Very large**)
 - Variance: $\approx \frac{n}{p}$ (**Very small**)
- **Empirically, we see “double descent” in deep neural networks:**

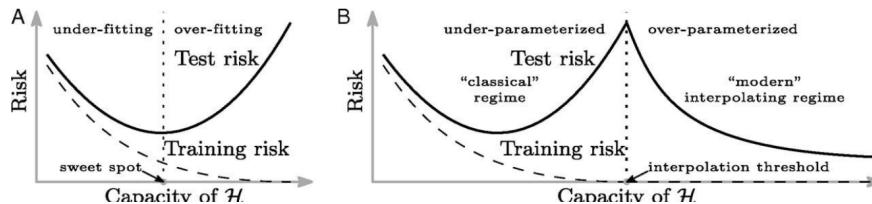


Figure 25: High-dimensional regime: the relationship between model complexity and error follows different dynamics. Bias dominates, and adding data helps only marginally when p is very large.

Interpretation:

- Adding more data (n) helps only marginally when p is very large
- The minimum-norm solution has low variance but high bias
- This is the opposite of the low-dimensional regime where variance dominates

Comparing Regimes

	Low-dim ($p \ll n$)	High-dim ($p \gg n$)
Risk scaling	$\frac{p}{n}\sigma^2$	$(1 - \frac{n}{p})\ \beta^*\ ^2 + \frac{n}{p}\sigma^2$
Dominant term	Variance	Bias
Effect of more data	Rapid improvement ($\propto 1/n$)	Marginal improvement
Key ratio	p/n	n/p

43.3 The Interpolation Threshold and Double Descent

Something interesting happens around $p \approx n$ —the **interpolation threshold**. This is where the model transitions from underfitting (not enough parameters to fit the data) to interpolating (enough parameters to fit exactly).

Double Descent Phenomenon

Classical U-shaped bias-variance curves predict that test error should increase monotonically as model complexity exceeds the optimal point. However, empirical observations with neural networks and other overparameterised models show **double descent**:

1. Error increases as p approaches n (classical regime)
2. Error **peaks** at the interpolation threshold $p \approx n$
3. Error **decreases again** as $p \gg n$ (overparameterised regime)

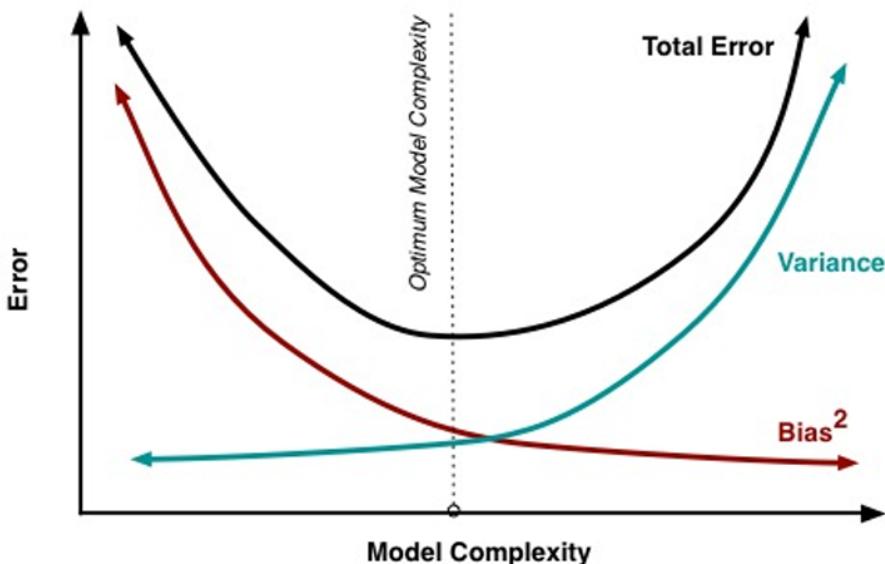


Figure 26: Classical bias-variance tradeoff: total error (MSE, solid black) is the sum of squared bias (decreasing with complexity) and variance (increasing with complexity). The optimal complexity balances these two components. Double descent extends this picture into the overparameterised regime, where error can decrease again.

Why does the peak occur at $p \approx n$? At the interpolation threshold:

- The model *just barely* has enough parameters to fit the training data

- Small changes in data cause large changes in the fitted model
- Variance is extremely high because the solution is sensitive to every training point
- This is the worst of both worlds: interpolating noise with high variance

Why does error decrease in the overparameterised regime? When $p \gg n$:

- Many different parameter vectors can interpolate the data
- The minimum-norm solution selects the “simplest” interpolator
- This implicit regularisation keeps the solution smooth despite perfect training fit
- As p increases, the minimum-norm solution becomes even smoother

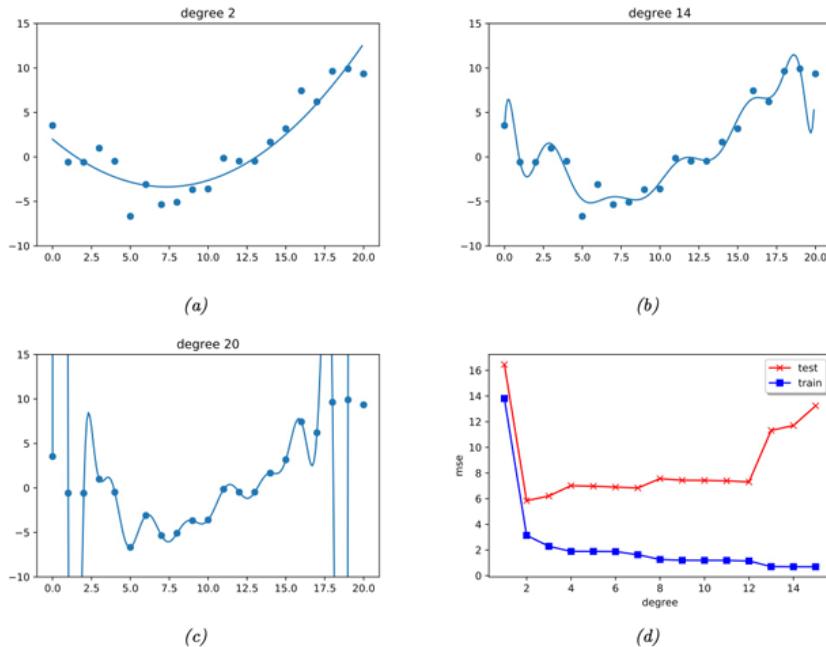


Figure 27: Effect of model complexity on fit. (a) Low complexity (e.g., degree 2 polynomial): underfitting, high bias—the model cannot capture the pattern. (b) Medium complexity (e.g., degree 14): good fit—balancing bias and variance. (c) High complexity (e.g., degree 20): classical overfitting, high variance. (d) Training error decreases monotonically with complexity; test error is U-shaped in the classical regime but can exhibit double descent in the overparameterised regime.

NB!

[Implications for Deep Learning] Double descent helps explain why deep neural networks with millions of parameters can generalise well despite classical theory predicting massive overfitting:

- They operate deep in the overparameterised regime ($p \gg n$)
- Gradient descent provides implicit regularisation toward “simpler” solutions
- The effective complexity is much lower than the parameter count suggests

However, this does **not** mean “more parameters is always better”:

- The peak at the interpolation threshold is real and can be severe
- Implicit regularisation depends on the optimisation algorithm and architecture
- Explicit regularisation (weight decay, dropout) is still beneficial

Connection to Week 5

Week 5 develops the theory of **benign overfitting**, explaining when and why interpolating models can generalise well:

- The **manifold hypothesis**: data has low intrinsic dimension even in high-dimensional spaces
- **k -split analysis**: SVD separates signal from noise components
- **Conditions for benign overfitting**: requires high-dimensional noise to be roughly isotropic

This provides a modern perspective complementing classical learning theory.

44 Bias-Variance Decomposition

The expected prediction error can be decomposed into interpretable components, providing fundamental insight into the sources of error.

Bias-Variance-Noise Decomposition

For squared error loss, the expected prediction error at a new point x can be written:

$$\mathbb{E}[(y - \hat{f}(x))^2] = \underbrace{\text{Bias}[\hat{f}(x)]^2}_{\text{Systematic error}} + \underbrace{\text{Var}[\hat{f}(x)]}_{\text{Estimation variability}} + \underbrace{\sigma^2}_{\text{Irreducible noise}}$$

where:

- $\text{Bias}[\hat{f}(x)] = \mathbb{E}[\hat{f}(x)] - f^*(x)$ measures systematic deviation from the truth
- $\text{Var}[\hat{f}(x)] = \mathbb{E}[(\hat{f}(x) - \mathbb{E}[\hat{f}(x)])^2]$ measures sensitivity to training data
- σ^2 is the inherent noise in the data-generating process

Unpacking each term:

Bias: If we could train our model on infinitely many datasets from the same distribution, bias is the difference between the average prediction and the truth. High bias means the model is systematically wrong, usually because it's too simple to capture the true pattern.

Variance: How much does the prediction change when we train on different datasets? High variance means the model is very sensitive to the particular training set—it's picking up on noise rather than signal.

Irreducible error: Even with the perfect model and infinite data, we can't do better than σ^2 because of inherent noise in y .

The fundamental tradeoff:

- **Simple models** (few parameters, strong regularisation): High bias, low variance. They consistently make similar predictions regardless of training data, but those predictions may be systematically off.
- **Complex models** (many parameters, weak regularisation): Low bias, high variance. They can capture complex patterns but are sensitive to the particular training set.

Bias-Variance in Different Regimes

Underparameterised ($p < n$):

- Increasing complexity: bias \downarrow , variance \uparrow
- Optimal complexity balances the two
- Classical U-shaped test error curve

At interpolation threshold ($p \approx n$):

- Variance can become very large
- Small changes in data cause large changes in the fit
- Often the worst point for generalisation

Overparameterised ($p \gg n$):

- With implicit regularisation: both bias and variance can be low
- This is the “benign overfitting” regime
- The minimum-norm solution provides regularisation

45 Summary

Key Concepts from Week 4

1. **Cross-validation:** The practical workhorse for estimating generalisation error
 - K-fold CV balances bias and variance of the estimate ($K = 10$ is standard)
 - LOOCV has a closed-form shortcut for linear regression via the hat matrix
 - One standard error rule implements Occam's razor—prefer simpler models within noise margin
 - Use grouped/time-series variants for non-i.i.d. data to prevent information leakage
 - Nested CV provides unbiased estimation when also tuning hyperparameters
2. **Model selection criteria:** AIC and BIC as alternatives to CV
 - AIC: targets prediction, penalty $2k$, tends to select larger models
 - BIC: targets model identification, penalty $k \log n$, tends to select smaller models
 - Both require likelihood framework; CV is more flexible
3. **Generalisation bounds:** Theoretical guarantees combining concentration inequalities (Hoeffding) and union bounds
 - Error scales with $|\mathcal{H}|$ or VC dimension
 - Error decreases with sample size n
 - Bounds are typically loose but provide conceptual insight
4. **VC dimension:** Measures hypothesis class complexity via shattering
 - Does not always equal parameter count ($\sin(\omega x)$ is the classic counterexample)
 - Enables bounds for infinite hypothesis classes
 - For linear classifiers in \mathbb{R}^d : $VC = d + 1$
5. **Rademacher complexity:** Data-dependent complexity measure
 - Measures ability to fit random labels
 - Can give tighter bounds than VC dimension for specific data distributions
6. **Dimensional regimes:** Low-dim ($p \ll n$) and high-dim ($p \gg n$) have different error dynamics
 - Low-dim: variance dominates, more data helps rapidly ($\propto 1/n$)
 - High-dim: bias dominates, more data helps marginally
 - Double descent: error peaks at $p \approx n$, then decreases in overparameterised regime
7. **Bias-variance tradeoff:** Fundamental decomposition of prediction error
 - Simple models: high bias, low variance
 - Complex models: low bias, high variance
 - Overparameterised models with implicit regularisation: can achieve low bias *and* low variance

46 The Overfitting Paradox

Chapter Summary

Classical statistical wisdom holds that models interpolating training data (zero training error) are badly overfit. Yet modern neural networks routinely interpolate and still generalise well. This chapter develops the theory of **benign overfitting**, explaining when and why this apparent paradox occurs. Key concepts: the **double descent** phenomenon challenges U-shaped bias-variance curves; **minimum-norm interpolation** provides implicit regularisation; and benign overfitting requires specific data structure—low-dimensional signal embedded in high-dimensional isotropic noise.

46.1 Classical Wisdom: Interpolation is Bad

The classical machine learning narrative, developed throughout the 20th century, tells a clear story about model complexity. This story has been the foundation of statistical learning theory for decades, and understanding it deeply is essential before we can appreciate why modern practice seems to violate it.

- **Underfitting:** Too simple a model cannot capture the underlying pattern. A linear model trying to fit quadratic data will systematically miss the curvature, no matter how much data we collect.
- **Optimal:** The “sweet spot” balances approximation error (bias from model misspecification) and estimation error (variance from finite samples). This is the regime where classical statistics excels.
- **Overfitting:** Too complex a model memorises noise, failing to generalise. A high-degree polynomial will pass through every training point but oscillate wildly between them, producing terrible predictions on new data.

This narrative is formalised in the bias-variance tradeoff (see Week 3): as model complexity increases, bias decreases but variance increases. The optimal model minimises their sum.

Classical Generalisation Bounds

Traditional learning theory bounds test error in terms of hypothesis class complexity. For a finite hypothesis class \mathcal{H} , with probability at least $1 - \delta$:

$$R(\hat{f}) \leq \hat{R}(\hat{f}) + \sqrt{\frac{\log |\mathcal{H}| + \log(1/\delta)}{2n}}$$

For infinite classes, complexity is measured via VC dimension or Rademacher complexity (Week 4). These bounds suggest test error grows with model complexity—a model with enough parameters to interpolate training data should generalise poorly.

Unpacking the classical bound: The bound above has two terms:

- $\hat{R}(\hat{f})$ is the **empirical risk**—the average loss on training data. An interpolating model has $\hat{R}(\hat{f}) = 0$.
- The second term is the **generalisation gap**—how much worse we expect to do on new data. This term grows with $\log |\mathcal{H}|$, the complexity of the hypothesis class.

The logic seems airtight: if your hypothesis class is rich enough to interpolate any training set (including noise), then $|\mathcal{H}|$ must be enormous, making the generalisation gap large. Even with zero training error, test error should be high.

The ultimate overfitting, by this view, is **interpolation**: fitting training data exactly, achieving zero training error. A model that memorises every training point, including noise, should perform terribly on new data.

46.2 Modern Observation: Deep Networks Interpolate and Generalise

Yet modern deep learning contradicts this story. Consider the empirical facts:

- Large neural networks have millions or billions of parameters—far more than training examples
- They routinely achieve zero or near-zero training error
- They *still* generalise well to unseen data
- Adding more parameters often *improves* test performance, not worsens it

NB!

The puzzle: Classical theory predicts that interpolating models should fail catastrophically. Modern practice shows they can succeed spectacularly. Either classical theory is wrong, or something subtle is happening that the classical view misses.

This is not a minor discrepancy—the gap between theory and practice is enormous. Classical bounds suggest networks with billions of parameters should have generalisation gaps measured in the hundreds or thousands of percentage points. Yet ImageNet classification works.

This chapter reconciles these perspectives. The resolution: **not all interpolating solutions are equal**. When there are infinitely many ways to fit training data exactly, the particular interpolating solution chosen by gradient descent (or equivalently, the minimum-norm solution) can have excellent generalisation properties—under specific conditions on the data.

47 Recap: OLS in Different Regimes

Before diving into benign overfitting, we need to understand why generalisation behaviour depends so critically on the relationship between p (number of features) and n (number of observations). This foundational analysis will reveal the mathematical structure that makes benign overfitting possible.

Relationship Between Risk, Loss, and Bias-Variance

Recall the key relationships from earlier weeks:

- **Risk** = expected loss over the data distribution: $R(f) = \mathbb{E}[\ell(f(X), Y)]$
- The **loss function** determines the form of risk
- When we use MSE as our risk (i.e., squared error loss), the risk decomposes nicely into bias and variance terms
- This decomposition is specific to squared error—other loss functions may not yield such clean decompositions

Why this matters: The bias-variance decomposition is not just a mathematical curiosity—it reveals the fundamental tension in statistical learning. Understanding how this tension manifests differently in low and high dimensions is the key to understanding benign overfitting.

47.1 Low-Dimensional Regime: $p \ll n$

When we have many more observations than predictors, OLS is in its “comfort zone.” This is the classical statistics setting where the Gauss-Markov theorem guarantees optimality.

Risk in Low Dimensions

Setting: $p \ll n$ (many more observations than features)

Consider the linear model $y = X\beta^* + \epsilon$ with $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$.

Risk formula:

$$R(\hat{\beta}) - R(\beta^*) = \frac{p}{n} \sigma^2 \quad (15)$$

Properties:

- OLS is BLUE (Best Linear Unbiased Estimator) via the Gauss-Markov theorem
- Risk decreases rapidly as n increases (proportional to $1/n$)
- Adding data helps significantly
- The estimator is unbiased: $\mathbb{E}[\hat{\beta}] = \beta^*$

Unpacking the low-dimensional risk formula: The formula $R(\hat{\beta}) - R(\beta^*) = \frac{p}{n} \sigma^2$ tells us several things:

- **Linear in p :** Each additional feature adds σ^2/n to the risk. More features means more parameters to estimate, each contributing uncertainty.
- **Inverse in n :** More data reduces risk proportionally. Doubling your sample size halves your excess risk—this is the parametric rate.
- **Proportional to σ^2 :** Noisier data means worse estimation. If labels are inherently unpredictable, even infinite data cannot achieve zero excess risk.
- **No bias term:** Because OLS is unbiased in this regime, the entire excess risk comes from variance—the randomness in estimating β^* from finite samples.

NB!

The formula $R(\hat{\beta}) - R(\beta^*) = \frac{p}{n} \sigma^2$ assumes that the SVD structure of the training data X matches that of test data \tilde{X} . This means the geometry (variance-covariance structure) is consistent between training and test sets. This is an idealised assumption—in practice, some regularisation or careful validation is necessary to ensure the theoretical rate of error reduction actually holds.

More fundamentally, this formula assumes $p < n$ so that $(X^\top X)^{-1}$ exists. At the boundary $p = n$, this inverse becomes singular, and the formula breaks down.

47.2 High-Dimensional Regime: $p \gg n$

This scenario is increasingly common in modern machine learning, where feature dimensionality often vastly exceeds sample size (genomics, image processing, text analysis). The behaviour here is qualitatively different from the low-dimensional case.

Risk in High Dimensions

Setting: $p \gg n$ (many more features than observations)

Risk formula:

$$R(\hat{\beta}) - R(\beta^*) \approx \left(1 - \frac{n}{p}\right) \|\beta^*\|^2 + \frac{n}{p} \sigma^2 \quad (16)$$

This decomposes into:

- **Bias term:** $\approx (1 - \frac{n}{p}) \|\beta^*\|^2$ — very large when $p \gg n$
- **Variance term:** $\approx \frac{n}{p} \sigma^2$ — very small when $p \gg n$

Unpacking the high-dimensional risk formula: This formula reveals a complete reversal of the low-dimensional picture:

The bias term $(1 - n/p)\|\beta^*\|^2$ dominates because:

- When $p \gg n$, the ratio $n/p \approx 0$, so $(1 - n/p) \approx 1$
- The model has infinitely many solutions that interpolate the data
- The minimum-norm solution is biased toward zero in directions the data doesn't constrain
- This bias depends on $\|\beta^*\|^2$ —larger true coefficients mean larger potential bias
- **Geometric intuition:** The data only constrains β in n directions. In the remaining $p - n$ directions, the minimum-norm solution sets coefficients to zero, but β^* may have non-zero components there.

The variance term $\frac{n}{p} \sigma^2$ is small because:

- The minimum-norm constraint severely limits which solutions are possible
- With so many parameters “explaining” relatively few observations, the estimates are tightly constrained
- Counterintuitively, having more parameters leads to *lower* variance (but much higher bias)
- **Analogy:** If you have 10 equations and 1000 unknowns, the minimum-norm solution is highly determined—there’s only one way to satisfy the equations with minimal norm

On the Margin, Sample Size Does Not Help Much

In the high-dimensional regime, marginally increasing n provides little benefit:

- The approximation $(1 - n/p)$ shows that when p is large, even substantial increases in n barely change the dominant bias term
- Going from $n = 100$ to $n = 200$ when $p = 10,000$ only reduces $(1 - n/p)$ from 0.99 to 0.98
- Prediction error does not significantly improve with more data because model complexity is too high relative to available information
- This is a fundamental limitation of unregularised OLS in high dimensions

47.3 Comparing the Two Regimes

Low vs High Dimensional OLS

	Low-dim ($p \ll n$)	High-dim ($p \gg n$)
Risk formula	$\frac{p}{n}\sigma^2$	$(1 - \frac{n}{p})\ \beta^*\ ^2 + \frac{n}{p}\sigma^2$
Dominant component	Variance	Bias
Effect of more data	Rapid improvement	Marginal improvement
Effect of more features	Risk increases	Risk may decrease!
OLS status	BLUE (optimal)	Problematic without regularisation

The high-dimensional formula contains a surprising prediction: as $p \rightarrow \infty$ with n fixed, the bias term $(1 - n/p)\|\beta^*\|^2 \rightarrow \|\beta^*\|^2$ while the variance term $n\sigma^2/p \rightarrow 0$. The excess risk approaches $\|\beta^*\|^2$, which is *finite*—not catastrophically large.

But what happens in between? At the transition $p \approx n$, something dramatic occurs.

47.4 The Regularisation Paradox

Implications for Regularisation

The behaviour in different regimes has led to the development of regularisation techniques:

- In low dimensions ($p \ll n$), OLS generally performs well with risk decreasing rapidly as data increases
- In high dimensions ($p \gg n$), traditional OLS struggles due to large bias and limited benefit from additional data
- This motivates techniques like Ridge regression and Lasso that deliberately introduce bias to reduce variance

NB!

The apparent paradox: In high dimensions, OLS already has high bias and low variance. Why would introducing *more* bias via regularisation help?

The resolution: The bias-variance decomposition above assumes using the minimum-norm OLS solution. Regularisation changes *which* solution we find, potentially trading a different kind of bias for better overall performance. Ridge regression, for instance, shrinks coefficients toward zero, which can reduce the effective complexity of the model and improve generalisation even though it technically adds bias.

The key insight is that not all bias is equally harmful—structured bias (like shrinkage toward zero) can be much less damaging than the unstructured bias of minimum-norm interpolation. Ridge regression’s bias is “aligned” with typical β^* structure (small coefficients), whereas minimum-norm bias is “aligned” with the null space of X (which may not correspond to small β^* components).

48 The Double Descent Phenomenon

Section Summary

The **double descent** curve reveals three regimes: (1) the classical U-shaped region where test error first decreases then increases with complexity, (2) a sharp peak at the **interpolation threshold** ($p \approx n$), and (3) a second descent where test error *decreases* in the overparameterised regime ($p \gg n$). This challenges the classical view that more parameters always risk overfitting.

Classical learning theory predicts that test error should increase monotonically with model complexity beyond the “sweet spot” where bias and variance are optimally balanced. But empirically, something surprising happens with highly overparameterised models.

48.1 From U-Curve to Double Descent

Classical theory predicts a U-shaped test error curve: error decreases as model complexity increases (reducing bias), reaches a minimum, then increases (as variance dominates).

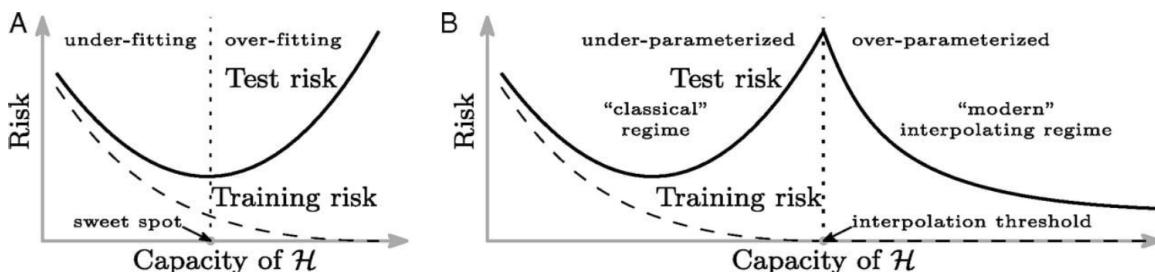


Figure 28: Double descent: test error peaks at the interpolation threshold ($p \approx n$), then **decreases** as $p \gg n$. The classical U-curve captures only the left portion of this phenomenon.

The modern picture extends this dramatically. As we push model complexity beyond the interpolation threshold:

1. **Underparameterised regime ($p < n$)**: Classical bias-variance tradeoff applies. Test error follows the familiar U-curve. More parameters initially help (reducing bias) but eventually hurt (increasing variance).

2. **Interpolation threshold** ($p \approx n$): The model can *just barely* fit training data exactly. Test error spikes dramatically. This is the worst possible regime.
3. **Overparameterised regime** ($p > n$): Many solutions interpolate the data. Among these, the minimum-norm solution generalises increasingly well as p grows. Test error *decreases*.

48.2 What Happens at the Interpolation Threshold?

The spike at $p \approx n$ is not merely a theoretical curiosity—it has a clear mathematical explanation and represents a genuine danger zone for practitioners.

The Interpolation Threshold Singularity

When $p = n$ exactly (assuming X has full rank):

- The system $X\beta = y$ has a unique solution
- This solution interpolates the training data exactly
- But the solution is **maximally sensitive** to noise

Mathematically, $(X^\top X)^{-1}$ has eigenvalues that approach infinity as we approach the threshold. Small perturbations in y (from noise) cause enormous changes in $\hat{\beta}$.

The condition number $\kappa(X^\top X) = \lambda_{\max}/\lambda_{\min}$ diverges as the smallest singular value of X approaches zero.

Unpacking the singularity: To understand why $p \approx n$ is so dangerous, consider what happens as we approach this threshold:

- **From below** ($p < n$): As p approaches n , we're adding more and more flexibility to the model. The matrix $X^\top X$ remains invertible, but its smallest eigenvalue shrinks toward zero. The inverse $(X^\top X)^{-1}$ has eigenvalues that grow without bound.
- **At $p = n$:** If X has full rank, there's exactly one solution. But this solution is found by inverting a nearly-singular matrix, amplifying any noise in y enormously.
- **From above** ($p > n$): Now $X^\top X$ is truly singular (rank $n < p$). We can't invert it directly, but there are infinitely many solutions. The minimum-norm solution turns out to be much better behaved than the unique solution at $p = n$.

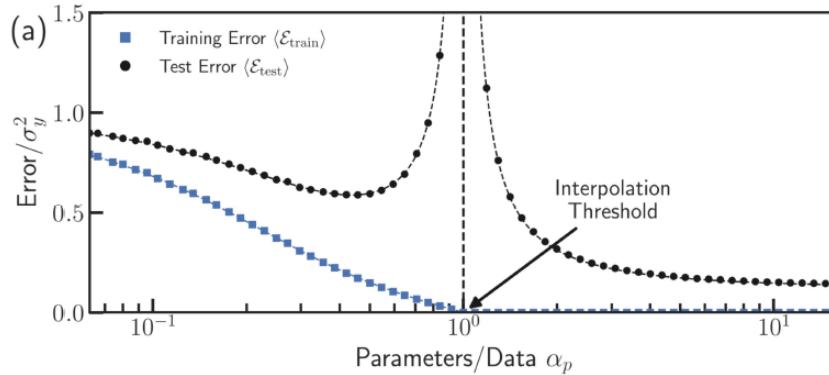


Figure 29: At $p \approx n$, the model can perfectly interpolate training data (zero training error), but does so in the worst possible way—with maximum sensitivity to noise.

Intuition: At the interpolation threshold, the model has exactly enough capacity to memorise the training data—no more, no less. It must contort itself maximally to pass through every training point. There’s no “slack” to smooth out noise.

Analogy: Imagine fitting a polynomial of degree $n - 1$ to n data points. The polynomial is forced to oscillate wildly to hit every point exactly. Now imagine fitting a polynomial of degree $10n$ to the same n points. There are infinitely many such polynomials, and we can choose one that interpolates smoothly without wild oscillations.

48.3 Why Does Error Decrease Again?

The key insight: once $p > n$, infinitely many solutions interpolate the training data. We’re no longer forced to use the unique, ill-conditioned solution.

Multiple Interpolating Solutions

When $p > n$, the system $X\beta = y$ is underdetermined. The solution set is an affine subspace:

$$\{\beta : X\beta = y\} = \hat{\beta}_{\text{particular}} + \text{null}(X)$$

Any β of the form $\hat{\beta} + v$ where $v \in \text{null}(X)$ also interpolates the data. Among these infinitely many solutions, which should we choose?

Unpacking the solution space: The null space of X , denoted $\text{null}(X)$, consists of all vectors v such that $Xv = 0$. When $p > n$, this null space has dimension $p - n > 0$.

- Any particular solution $\hat{\beta}_{\text{particular}}$ satisfies $X\hat{\beta}_{\text{particular}} = y$
- Adding any $v \in \text{null}(X)$ gives another solution: $X(\hat{\beta}_{\text{particular}} + v) = X\hat{\beta}_{\text{particular}} + Xv = y + 0 = y$
- The solution space is thus an infinite $(p - n)$ -dimensional affine subspace
- Different solutions in this space can have vastly different generalisation properties

The answer—and the key to benign overfitting—is the **minimum-norm solution**.

49 Minimum-Norm Interpolation

Section Summary

When infinitely many solutions interpolate the data, the **minimum-norm** solution—the interpolating $\hat{\beta}$ with smallest $\|\beta\|_2$ —has special properties. It acts as **implicit regularisation**, avoiding solutions that amplify noise. This is the solution gradient descent converges to, and it can be computed via the pseudoinverse.

49.1 Definition and Motivation

Minimum-Norm Interpolation

The **minimum-norm interpolating solution** is:

$$\hat{\beta}_{\min\text{-norm}} = \arg \min_{\beta} \|\beta\|_2 \quad \text{subject to } X\beta = y$$

This is the solution with smallest Euclidean norm among all solutions that perfectly fit the training data.

Unpacking the definition: The minimum-norm solution is a constrained optimisation problem:

- **Constraint:** $X\beta = y$ — the solution must interpolate the training data perfectly
- **Objective:** $\min \|\beta\|_2$ — among all interpolating solutions, choose the one with smallest length
- This is a convex optimisation problem with a unique solution (the norm is strictly convex)

Why minimum norm? Several complementary perspectives illuminate why this is the “right” choice:

1. **Occam’s razor:** Among all interpolating solutions, prefer the “simplest”—the one requiring the smallest coefficients. Large coefficients suggest the model is doing something extreme; small coefficients suggest a more moderate, robust fit.
2. **Implicit regularisation:** Large coefficients typically indicate overfitting. Minimum norm avoids this without explicit penalties. It’s as if we’re applying L^2 regularisation, but with the regularisation strength automatically tuned to the minimum value that still allows interpolation.
3. **Gradient descent:** When training overparameterised linear models with gradient descent starting from $\beta_0 = 0$, the algorithm converges to the minimum-norm solution. This is not a coincidence—gradient descent naturally finds the solution closest to its initialisation.
4. **Ridge limit:** As $\lambda \rightarrow 0^+$, ridge regression $\hat{\beta}_\lambda = (X^\top X + \lambda I)^{-1} X^\top y$ converges to the minimum-norm interpolating solution. The regularisation parameter “selects” the minimum-norm solution in the limit.

Connection to Ridge Regression

For $p > n$, consider the ridge estimator with vanishing regularisation:

$$\lim_{\lambda \rightarrow 0^+} (X^\top X + \lambda I)^{-1} X^\top y = X^+ y = \hat{\beta}_{\text{min-norm}}$$

where X^+ is the Moore-Penrose pseudoinverse. Ridge regression with $\lambda > 0$ gives a unique, well-defined solution. As $\lambda \rightarrow 0$, this solution approaches the minimum-norm interpolating solution.

Key insight: The limit exists and is well-behaved, even though $(X^\top X)^{-1}$ doesn't exist when $p > n$. The regularisation "selects" the minimum-norm solution from the infinite set of interpolators.

Unpacking the ridge connection: This connection is mathematically beautiful and practically important:

- When $p > n$, $(X^\top X)$ is singular—it has zero eigenvalues corresponding to the null space of X
- Adding λI shifts all eigenvalues up by λ , making the matrix invertible
- As $\lambda \rightarrow 0$, the solution smoothly approaches the minimum-norm interpolator
- This provides a computational path to the minimum-norm solution and connects it to the well-understood theory of ridge regression

49.2 Why Minimum Norm Helps Generalisation

Consider what minimum norm does geometrically:

Geometric Intuition

The minimum-norm solution projects y onto the row space of X , then finds the corresponding β :

$$\hat{\beta}_{\text{min-norm}} = X^\top (X X^\top)^{-1} y$$

This solution:

1. Lives entirely in the row space of X (no component in the null space)
2. Distributes the "work" of fitting y across all features
3. Avoids placing large weights on arbitrary directions

Unpacking the geometry: The row space and null space of X are orthogonal complements in \mathbb{R}^p . Any vector β can be decomposed as:

$$\beta = \beta_{\text{row}} + \beta_{\text{null}}$$

where β_{row} is in the row space of X and β_{null} is in the null space.

The minimum-norm solution has $\beta_{\text{null}} = 0$. Why does this help?

- **Null space components don't affect training predictions:** If $v \in \text{null}(X)$, then $Xv = 0$, so adding v to β doesn't change $X\beta$.

- **But they can affect test predictions:** For new data \tilde{x} , we predict $\tilde{x}^\top \beta$. If β has null-space components, these contribute to the prediction even though they weren't constrained by training data.
- **Null-space components are “free” during training:** They can take any value without affecting training error. An optimisation process that doesn't constrain them might pick arbitrary, harmful values.

The null space of X represents directions in β -space that don't affect training predictions. Adding any $v \in \text{null}(X)$ to $\hat{\beta}$ doesn't change $X\hat{\beta}$, so training error remains zero. But these null-space components can dramatically affect test predictions on new data.

NB!

A non-minimum-norm interpolating solution has arbitrary components in the null space. These components don't help fit training data, but they can catastrophically hurt test performance by assigning large weights to “noise directions” that happen to correlate with new test points.

The minimum-norm solution, having zero null-space component, avoids this pathology entirely.

Example: Suppose X has a null-space direction v that happens to correlate strongly with some test points. A solution with a large component along v will make systematically wrong predictions on those test points, even though v was invisible during training.

50 SVD Perspective on Overparameterised Regression

Section Summary

The singular value decomposition (SVD) provides the clearest mathematical framework for understanding benign overfitting. It reveals how the design matrix X naturally separates signal (large singular values) from noise (small singular values), and shows exactly how the minimum-norm solution achieves implicit regularisation.

50.1 SVD Basics Revisited

When $p > n$, $X^\top X$ is singular and we cannot compute $(X^\top X)^{-1}$. The SVD provides an alternative that is both computationally tractable and theoretically illuminating.

SVD of the Design Matrix

Any $n \times p$ matrix X of rank $r \leq \min(n, p)$ can be decomposed as:

$$X = U\Sigma V^\top$$

where:

- U is $n \times r$ with orthonormal columns: $U^\top U = I_r$. These are the **left singular vectors**.
- Σ is $r \times r$ diagonal with singular values $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > 0$
- V is $p \times r$ with orthonormal columns: $V^\top V = I_r$. These are the **right singular vectors**.

The columns of V form an orthonormal basis for the row space of X . The columns of U form an orthonormal basis for the column space.

Unpacking the SVD: The decomposition $X = U\Sigma V^\top$ reveals the fundamental structure of any linear map:

- V^\top **rotates** from the original p -dimensional space to an r -dimensional coordinate system aligned with the principal directions of X
- Σ **scales** each coordinate by the corresponding singular value—stretching or shrinking along each principal direction
- U **rotates** from this intermediate space to the output n -dimensional space

The singular values σ_i measure how much X “stretches” space in each principal direction. Large singular values indicate directions where the data varies substantially; small singular values indicate directions where the data is nearly constant.

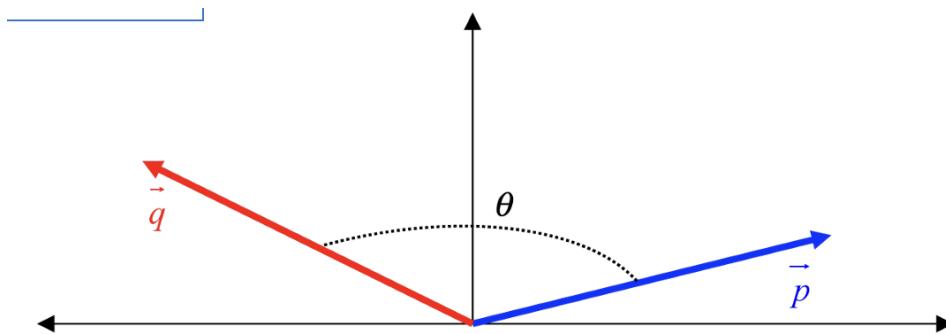


Figure 30: The SVD geometrically represents how a matrix transforms space: rotation, scaling along principal axes, then another rotation.

50.2 Minimum-Norm Solution via SVD

Minimum-Norm Solution via Pseudoinverse

The Moore-Penrose pseudoinverse of X is:

$$X^+ = V\Sigma^{-1}U^\top$$

The minimum-norm interpolating solution is:

$$\hat{\beta}_{\text{min-norm}} = X^+y = V\Sigma^{-1}U^\top y$$

Training predictions are:

$$\hat{y}_{\text{train}} = X\hat{\beta} = U\Sigma V^\top \cdot V\Sigma^{-1}U^\top y = UU^\top y$$

This is the projection of y onto the column space of X .

Unpacking the pseudoinverse: The formula $\hat{\beta} = V\Sigma^{-1}U^\top y$ has a clear interpretation:

1. $U^\top y$: Project y onto the column space of X , expressing it in the basis of left singular vectors. This gives r coefficients representing how much of y lies along each principal direction.
2. Σ^{-1} : Divide each coefficient by the corresponding singular value. This “undoes” the scaling that X performs. Directions where X stretches a lot (large σ_i) get divided by a large number; directions where X barely stretches (small σ_i) get divided by a small number.
3. V : Map back to the original p -dimensional coefficient space, using the right singular vectors as a basis.

SVD Interpretation

The SVD reveals the minimum-norm solution’s structure:

1. $U^\top y$ projects y onto the r principal directions of X
2. Σ^{-1} rescales by the inverse singular values
3. V maps back to coefficient space

Crucially, $\hat{\beta}$ lies in the column space of V (the row space of X). It has no component in $\text{null}(X)$.

NB!

A common source of confusion: in the formula $\hat{\beta} = V\Sigma^{-1}U^\top y$, you do not need to explicitly compute $\hat{\beta}$ to make predictions on the training data. The SVD provides a direct linear projection (UU^\top) onto X . For new data \tilde{X} , predictions are $\tilde{X}\hat{\beta} = \tilde{X}V\Sigma^{-1}U^\top y$.

Also note: Σ^{-1} divides by singular values. If some singular values are very small, this division amplifies noise. This is why the interpolation threshold ($p \approx n$ with X barely full rank) is dangerous—small singular values lead to large, noise-sensitive coefficients.

50.3 Signal vs Noise: The k -Split Perspective

The key insight for benign overfitting is that SVD naturally separates “signal” from “noise” in the feature space.

The k -Split Decomposition

Suppose the singular values of X naturally divide into two groups:

- **Large singular values** ($\sigma_1, \dots, \sigma_k$): Correspond to directions where data varies substantially—the “signal” directions
- **Small singular values** ($\sigma_{k+1}, \dots, \sigma_r$): Correspond to directions where data varies little—the “noise” directions

We can write:

$$X = U_k \Sigma_k V_k^\top + U_{k:r} \Sigma_{k:r} V_{k:r}^\top$$

where the first term captures the dominant k directions and the second captures the remaining $r - k$ directions.

Correspondingly, we can conceptually split:

$$\beta^* = [\beta_{1:k}^*, \beta_{k+1:p}^*]$$

where $\beta_{1:k}^*$ represents the signal and $\beta_{k+1:p}^*$ represents coefficients in the noise subspace.

Unpacking the k -split: This decomposition is the mathematical foundation of benign overfitting theory:

- **Signal subspace** (first k directions): These directions have large singular values, meaning the data varies substantially along them. They contain the predictive information. The minimum-norm solution can estimate coefficients in these directions accurately because the data provides strong signal.
- **Noise subspace** (remaining $r - k$ directions): These directions have small singular values, meaning the data varies little along them. They contain mostly noise. The minimum-norm solution’s estimates in these directions are unreliable, but if there are many such directions and they’re isotropic (no preferred orientation), their errors average out.

The idea

- $\beta^* = [\beta_{0:k}^*, \beta_{k:\infty}^*]$
- $\beta_{0:k}^*$
 - The **low dimensional** part
 - This can behave like OLS when $p \ll n$
- $\beta_{k:\infty}^*$
 - The **high dimensional** part is assumed to be zero
 - This can be excluded as unimportant through the manifold hypothesis
 - It won't overly bias results because it behaves like white noise
- Critical: this split is *within the SVD*
 - Large singular values are in $\beta_{0:k}^*$, while small ones are in $\beta_{k:\infty}^*$

Figure 31: SVD naturally orders dimensions by importance. The first k singular values capture structure; the remainder behave like noise. Benign overfitting occurs when the “noise” dimensions are numerous and isotropic.

SVD as Automatic Feature Selection

SVD reorders features so that:

1. Each transformed feature (singular vector) is orthogonal to all others—no collinearity
2. Features are ordered by importance (singular value magnitude)
3. The first k features capture most of the variance in X
4. This is a **rotation** of the feature space that reveals intrinsic structure without changing distances

The SVD does not discard information—it reorganises it so that “importance” is monotonically decreasing across dimensions.

50.4 Two Perspectives on the k -Split

Formal Interpretations of the k -Split

Perspective 1: Coefficient decomposition (conceptual)

Splitting $\beta^* = [\beta_{1:k}^*, \beta_{k+1:p}^*]$ reflects:

- A prioritisation of features deemed most informative
- The $1 : k$ part resides in “effective low dimensionality” and can behave like OLS when $k \ll n$
- The $k + 1 : p$ part is relegated to noise

Perspective 2: SVD truncation (mechanistic)

Keeping only the first k singular values (and corresponding vectors) means:

- Approximating X by its most significant components
- Reducing effective model complexity
- Mitigating overfitting by disregarding dimensions that contribute little to variance

Key insight: The split is *within the SVD*—SVD is simultaneously enabling regression in high dimensions (bypassing singularity) and performing implicit dimensionality reduction.

51 Benign Overfitting: Formal Conditions

Section Summary

Benign overfitting occurs under specific conditions: the true signal must lie in a low-dimensional subspace, and the remaining high-dimensional “noise” directions must be approximately isotropic. When these conditions hold, the minimum-norm interpolator’s excess risk is controlled by the intrinsic dimension, not the ambient dimension.

51.1 Effective Rank: Measuring Spread

Before stating the conditions for benign overfitting, we need a way to measure how “spread out” the singular values are in the noise subspace.

Effective Rank

The **effective rank** of a covariance matrix Σ with eigenvalues $\lambda_1 \geq \lambda_2 \geq \dots$ is:

$$R(\Sigma) = \frac{(\sum_i \lambda_i)^2}{\sum_i \lambda_i^2} = \frac{\text{tr}(\Sigma)^2}{\|\Sigma\|_F^2}$$

This measures how many “effective dimensions” the data spans:

- If all eigenvalues are equal: $R(\Sigma) = p$ (full rank)
- If one eigenvalue dominates: $R(\Sigma) \approx 1$
- More spread eigenvalues \Rightarrow higher effective rank

For benign overfitting, we need the **tail effective rank**—the effective rank of the matrix restricted to the “noise” subspace (singular values beyond k).

Unpacking effective rank: The formula $R(\Sigma) = \frac{(\sum_i \lambda_i)^2}{\sum_i \lambda_i^2}$ is a ratio of squared norms:

- **Numerator** $(\sum_i \lambda_i)^2 = \text{tr}(\Sigma)^2$: The square of the total variance (sum of all eigenvalues)
- **Denominator** $\sum_i \lambda_i^2 = \|\Sigma\|_F^2$: The squared Frobenius norm, which weights each eigenvalue by its magnitude

Intuition: If eigenvalues are equal ($\lambda_i = c$ for all i), then $R(\Sigma) = (pc)^2/(pc^2) = p$. If one eigenvalue dominates ($\lambda_1 \gg \lambda_i$ for $i > 1$), then $R(\Sigma) \approx \lambda_1^2/\lambda_1^2 = 1$. The effective rank interpolates between these extremes.

51.2 The Risk Bound

Risk Bound for Benign Overfitting

Under appropriate conditions (sub-Gaussian design, bounded coefficients), the excess risk of the minimum-norm interpolator satisfies:

$$R(\hat{\beta}) - R(\beta^*) \lesssim \frac{\sigma^2}{c} \left(\frac{k^*}{n} + \frac{n}{R_{k^*}(\Sigma)} \right)$$

where:

- σ^2 : Noise variance in the response
- c : A constant depending on distributional assumptions
- k^* : Intrinsic dimension of the signal (number of “important” singular values)
- n : Sample size
- $R_{k^*}(\Sigma)$: Effective rank of the “tail” (noise) part of the covariance

Unpacking the bound: This formula is the central result of benign overfitting theory. Let’s examine each component:

- \lesssim : This notation means “bounded by up to constants”—the inequality holds up to multiplicative and additive factors that don’t depend on the key quantities.
- σ^2/c : An overall scale factor. More noise (σ^2) means higher risk. The constant c depends on how well-behaved the data distribution is.
- **Two additive terms:** The bound is the sum of two terms, corresponding to the signal and noise parts of the problem.

51.3 Understanding the Two Terms

The bound has two additive components, corresponding to the two parts of the k -split:

First term: k^*/n (**Classical parametric rate**)

- This is the familiar rate from low-dimensional OLS: risk $\propto p/n$
- Here, k^* plays the role of effective dimensionality
- The low-dimensional part of the model (first k^* singular directions) behaves like classical OLS with k^* features
- This term decreases as n grows, just as in standard regression
- **Compare to:** In low-dimensional OLS, risk is $\frac{p}{n}\sigma^2$; here the analogous term is $\frac{k^*}{n}$ (up to constants)

Second term: $n/R_{k^*}(\Sigma)$ (**Novel high-dimensional contribution**)

- This captures the contribution from the high-dimensional “noise” part
- $R_{k^*}(\Sigma)$ measures the “effective number of directions” in the high-dimensional subspace
- When $R_{k^*}(\Sigma)$ is large (many effective noise dimensions), this term is small
- The noise “averages out” across many dimensions rather than concentrating in a few
- This term can actually *decrease* as we add more features, if those features spread out the noise

When is Overfitting Benign?

The high-dimensional part doesn’t hurt generalisation when:

1. It spans **many different directions** (high effective rank $R_{k^*}(\Sigma)$)
2. It behaves like **isotropic noise**—random variation spread uniformly across many dimensions
3. The **effective dimension ratio** $n/R_{k^*}(\Sigma)$ is small

If the “noise dimensions” point in many different directions, they average out in predictions rather than systematically biasing the model.

51.4 Three Conditions for Benign Overfitting

Necessary Conditions

Benign overfitting requires:

1. Low intrinsic signal dimension

$$k^* \ll p$$

The true signal β^* lives primarily in a low-dimensional subspace. The first few singular directions capture the meaningful structure.

2. Isotropic noise in high dimensions

Eigenvalues $\lambda_{k^*+1}, \dots, \lambda_p$ are approximately equal

The remaining dimensions behave like isotropic Gaussian noise—no preferred direction in the noise subspace.

3. High effective rank in the tail

$$R_{k^*}(\Sigma) \gg n$$

The noise spreads across sufficiently many directions that it averages out. This requires p to be much larger than n in the noise subspace.

When all three conditions hold, the minimum-norm interpolator achieves risk comparable to the oracle estimator that knows k^* and projects onto the signal subspace.

Unpacking the conditions: Each condition plays a distinct role:

1. **Low intrinsic dimension** ensures the signal can be learned. If signal were spread across all p dimensions, we'd need $n \gg p$ to estimate it—the classical requirement. Low intrinsic dimension means we effectively have a k^* -dimensional problem.
2. **Isotropic noise** ensures no spurious patterns. If the noise had structure (e.g., concentrated in a few directions), the model might “learn” this structure and be misled. Isotropy means noise is direction-independent.
3. **High effective rank** ensures averaging. If noise were concentrated in a few directions, errors in those directions would dominate. High effective rank means errors are spread across many directions and tend to cancel.

NB!

The high-dimensional noise must behave like **isotropic** (white) noise for benign overfitting to work. If the noise has structure—correlations, preferred directions, low effective rank—it can systematically bias the model and destroy generalisation.

Example failure case: If the noise subspace has one dominant direction that happens to correlate with the test distribution, the minimum-norm solution will place large weight on this direction, causing poor generalisation.

Another failure case: If the noise dimensions are correlated (non-isotropic), the model might fit spurious patterns that don't generalise.

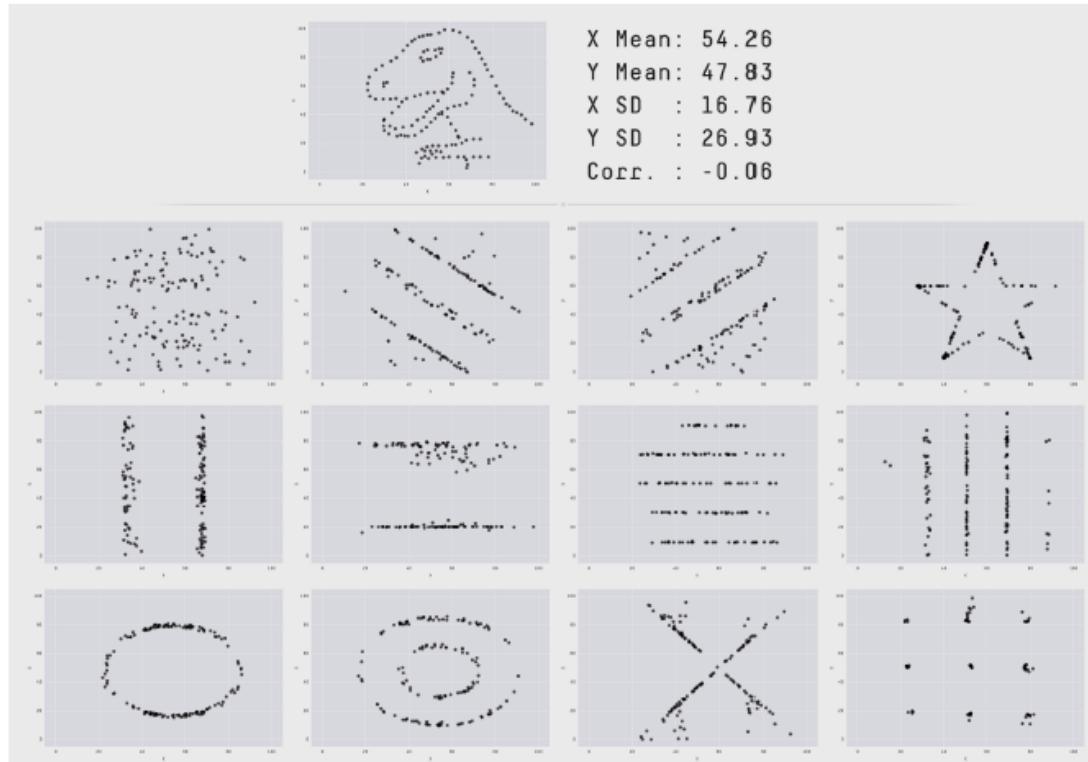


Figure 32: The structure of the covariance matrix matters. Isotropic noise (left) spreads errors across many dimensions; structured noise (right) can concentrate errors and break benign overfitting.

51.5 Why SVD Makes This Automatic

The minimum-norm solution computed via SVD automatically:

1. **Identifies signal directions:** The leading singular vectors capture where the data varies most
2. **Orders by importance:** Singular values quantify each direction's importance
3. **Distributes weight appropriately:** The pseudoinverse $V\Sigma^{-1}U^\top$ weights each direction inversely to its singular value, naturally downweighting noisy directions
4. **Stays in the row space:** By construction, $\hat{\beta}_{\min\text{-norm}}$ has no component in the null space of X

Implicit Regularisation via Minimum Norm

The minimum-norm constraint acts as **implicit regularisation**:

- It doesn't explicitly penalise complexity
- But among all interpolating solutions, it picks the one that doesn't amplify noise
- Directions with small singular values get small coefficients
- This achieves the effect of regularisation without an explicit penalty

This is why gradient descent on overparameterised models often works well: it tends to find minimum-norm solutions, which have this implicit regularisation property.

52 The Manifold Hypothesis

The mathematical conditions for benign overfitting connect to a broader principle about real-world data that explains why these conditions are often satisfied in practice.

The Manifold Hypothesis

High-dimensional real-world data typically lies near a **low-dimensional manifold**. The *intrinsic dimensionality* is much lower than the *ambient dimensionality*.

A manifold is a mathematical space that might locally look like flat Euclidean space but has more complex, curved structure globally.

Key concepts:

- **Ambient dimensionality:** The nominal dimension of the data space (e.g., number of pixels in an image)
- **Intrinsic dimensionality:** The true number of degrees of freedom needed to describe the data

Analogy: The surface of the Earth is a 2-dimensional manifold embedded in 3-dimensional space. Although we live in 3D, we only need two coordinates (latitude and longitude) to specify any location on the surface.

Unpacking the manifold hypothesis: This hypothesis has profound implications:

- **Ambient vs. intrinsic dimension:** An image might have millions of pixels (ambient dimension), but meaningful variations (pose, lighting, expression) span a much smaller space (intrinsic dimension). Most pixel configurations don't look like valid images.
- **Why manifolds:** Real data is generated by physical processes with limited degrees of freedom. A face image is constrained by anatomy, physics of light, camera optics—all of which reduce the effective dimensionality.
- **Consequences for learning:** If data lies on a low-dimensional manifold, we only need to learn a function on that manifold, not on the entire ambient space. This dramatically reduces the complexity of the learning problem.

Examples of Low Intrinsic Dimensionality

- **Face images:** Live in a space of millions of pixels, but meaningful variations (pose, lighting, expression, identity) span perhaps 50–100 dimensions
- **Natural images:** Despite high pixel counts, most random pixel configurations do not look like natural scenes—real images are constrained to a tiny subset of pixel space
- **Text:** The space of all possible character sequences is vast, but coherent text occupies a vanishingly small fraction
- **Genomic data:** Tens of thousands of genes, but biological states often involve coordinated changes in small gene modules
- **Physical systems:** State spaces are often constrained by conservation laws, symmetries, and physical constraints

d	Volume of domain of X	Volume of sphere	Fraction of data nearby	$\epsilon = \frac{1}{2}$
1	2^1	2ϵ	ϵ	$\frac{1}{2} = 0.5$
2	2^2	$\pi\epsilon^2$	$\frac{\pi}{4}\epsilon^2$	$\frac{\pi}{16} \approx 0.20$
3	2^3	$\frac{4\pi}{3}\epsilon^3$	$\frac{\pi}{6}\epsilon^3$	$\frac{\pi}{48} \approx 0.07$
4	2^4	$\frac{\pi^2}{2}\epsilon^4$	$\frac{\pi^2}{32}\epsilon^4$	$\frac{\pi^2}{512} \approx 0.02$

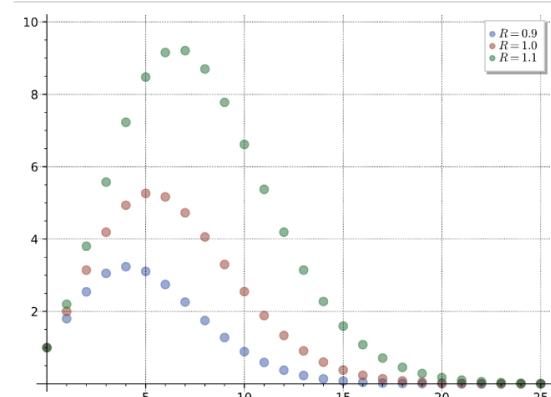


Figure 33: The curse of dimensionality: as dimension increases, volume concentrates near the surface of hyperspheres and hypercubes, making the interior essentially empty. Real data avoids this by lying on low-dimensional manifolds.

The manifold hypothesis explains why the conditions for benign overfitting are often satisfied in practice:

1. Real data has low intrinsic dimension $\Rightarrow k^*$ is small
2. The “extra” dimensions represent noise or irrelevant variation
3. This noise often has no preferred direction \Rightarrow approximately isotropic

Connection to double descent: Deep neural networks and overparameterised models are exceptionally good at discovering and exploiting low-dimensional structure within high-dimensional data. Through training:

- They learn to ignore irrelevant dimensions (noise)
- They focus on the manifold’s structure, capturing patterns that generalise
- The extra parameters provide flexibility to represent complex manifold geometry
- Implicit regularisation (from gradient descent, architecture choices) keeps solutions smooth

The underlying structure—not the nominal dimensionality—determines what can be learned. This is why complex models can generalise: they are not fitting to p independent dimensions, but to a much smaller intrinsic dimensionality.

53 Connection to Deep Learning

Section Summary

The theory of benign overfitting extends beyond linear models to help explain deep learning’s success. Neural networks benefit from overparameterisation through **implicit regularisation** in gradient descent. However, the full picture for deep learning remains an active research area—nonlinearity introduces both opportunities and challenges not captured by the linear theory.

53.1 Why Neural Networks Benefit from Overparameterisation

The benign overfitting phenomenon helps explain several puzzles in deep learning:

1. More parameters can improve generalisation

Classical theory predicts that adding parameters increases overfitting risk. But in practice, making neural networks wider or deeper often improves test performance, even when training error is already zero. The minimum-norm perspective suggests why: more parameters mean more interpolating solutions, and the one found by gradient descent may generalise better.

2. Gradient descent finds good solutions

For overparameterised networks trained from small random initialisation, gradient descent converges to the minimum-norm solution (in an appropriate sense). This provides implicit regularisation without explicit penalties.

Implicit Bias of Gradient Descent

For linear models, gradient descent starting from $\beta_0 = 0$ converges to the minimum-norm interpolator. For neural networks, the picture is more complex but analogous:

- Gradient descent with small initialisation favours “simple” functions
- The learned function has low complexity in a function-space sense
- Early stopping provides additional regularisation

This **implicit bias** toward simplicity helps explain why deep networks generalise despite their enormous capacity.

Unpacking implicit bias: The implicit bias of gradient descent is subtle but crucial:

- **For linear models:** Gradient descent from $\beta_0 = 0$ moves in the direction of the gradient, which lies in the row space of X . It never acquires null-space components, so it converges to the minimum-norm solution.
- **For neural networks:** The story is more complex because the loss landscape is non-convex. However, gradient descent tends to find “flat” minima (where the Hessian has small eigenvalues), which correspond to functions that are stable under perturbations—a form of simplicity.

- **Early stopping:** Even if gradient descent would eventually find a complex solution, stopping early keeps the solution close to initialisation, which tends to be simple.

3. Double descent in deep learning

Empirically, neural networks exhibit double descent in multiple ways:

- **Model-wise:** Test error peaks then decreases as width/depth increases
- **Epoch-wise:** Test error can show a second descent late in training
- **Sample-wise:** Adding training data can temporarily hurt then help

53.2 Limitations of the Linear Theory

The linear theory of benign overfitting provides valuable intuition but doesn't fully explain deep learning:

NB!

What the linear theory captures:

- Why interpolation isn't always bad
- How minimum-norm provides implicit regularisation
- The role of data geometry (manifold hypothesis)
- The importance of being well past the interpolation threshold

What it misses:

- How neural networks *learn* good representations (feature learning)
- The role of depth and hierarchical structure
- Why specific architectures (convolutions, attention) help
- Optimisation landscape effects (local minima, saddle points)
- The role of batch normalisation, dropout, and other techniques

For deep networks, the situation is richer: the network simultaneously learns features and fits the data. The benign overfitting perspective applies to the final linear layer, but feature learning introduces additional implicit biases that the linear theory cannot capture.

Connection to Deep Learning

Deep neural networks routinely have more parameters than training examples, yet generalise well. Benign overfitting theory helps explain this:

- Real-world data has low intrinsic dimensionality (manifold hypothesis)
- Networks learn to represent this low-dimensional structure
- “Noise” dimensions of the parameter space are handled via implicit regularisation
- Gradient descent finds solutions with beneficial properties (low norm, smooth)
- The interpolation threshold is avoided by being firmly in the overparameterised regime

54 Kernel Methods and Benign Overfitting

The theory of benign overfitting connects naturally to kernel methods, which provide a bridge between linear models and neural networks.

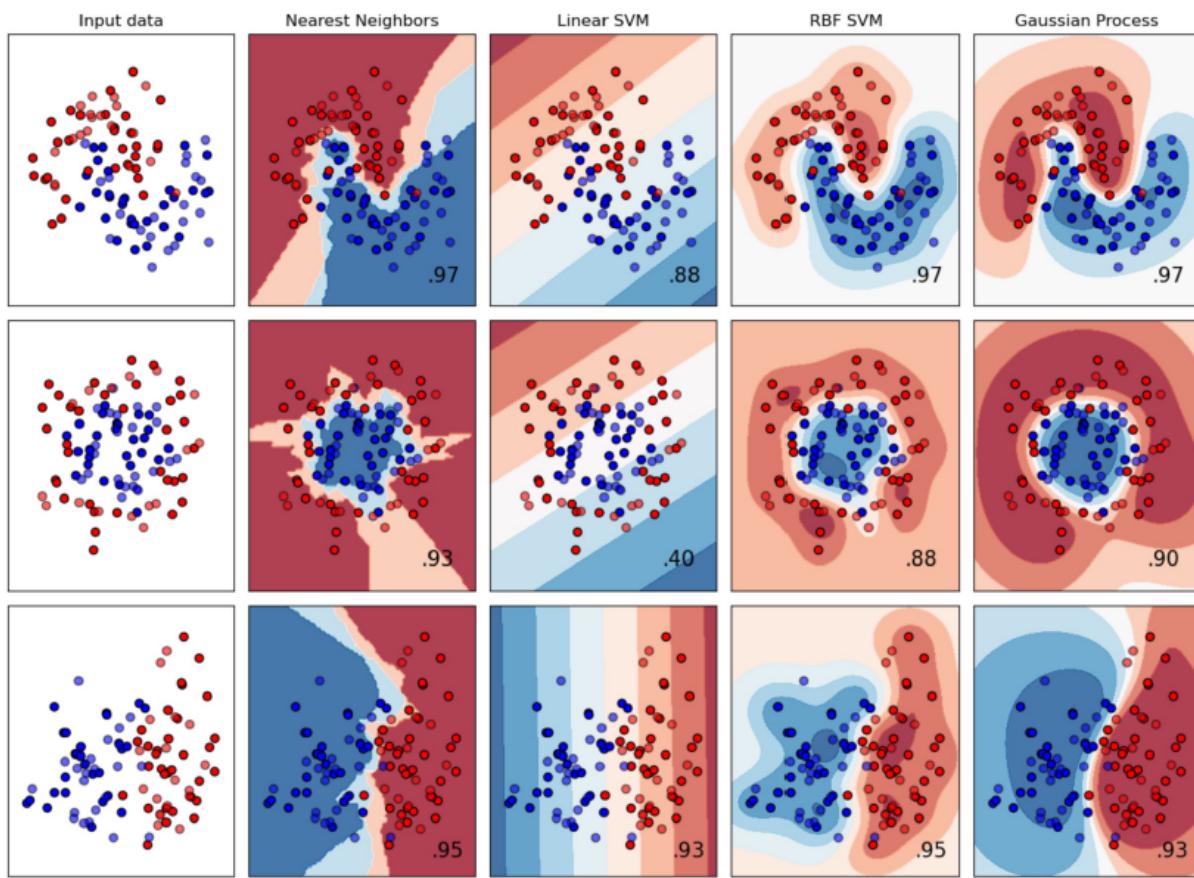


Figure 34: Different kernel choices correspond to different implicit feature spaces. The kernel determines the geometry of the function space and thus the implicit regularisation properties.

Kernels and Implicit Feature Spaces

A kernel $K(x, x')$ implicitly defines a feature map $\phi : \mathcal{X} \rightarrow \mathcal{H}$ such that $K(x, x') = \langle \phi(x), \phi(x') \rangle_{\mathcal{H}}$.

- The feature space \mathcal{H} can be infinite-dimensional (e.g., for the RBF kernel)
- Kernel regression finds the minimum-norm solution in this (possibly infinite-dimensional) feature space
- The kernel determines which functions are “simple” (low norm) and which are “complex” (high norm)

This connects to benign overfitting: kernel methods operate in high (or infinite) dimensional feature spaces, yet can generalise well because they find minimum-norm solutions.

Manipulating Kernels

allows you to construct complicated kernels

- Given two kernels $k_1(x, x')$ and $k_2(x, x')$, the following are all valid:

- | | |
|--|---|
| <ul style="list-style-type: none"> • $ck_1(x, x')$ • Multiply by a constant • $f(x)k_1(x, x')f(x')$ • Pre and multiply by some function of the inputs • $\text{poly}(k_1(x, x'))$ • Pass it through a polynomial • $\exp(k_1(x, x'))$ • Pass it through an exponential | <ul style="list-style-type: none"> • $k_1(x, x') + k_2(x, x')$ • Add two together • $k_1(x, x')k_2(x, x')$ • Multiply them together • $k_3(\phi(x), \phi(x'))$ • Expand feature and pass through a valid kernel • $x^T A x'$ • A bilinear form with the symmetric PSD matrix A |
|--|---|

Figure 35: Kernel manipulation: combining and transforming kernels creates new implicit feature spaces with different inductive biases.

The connection between kernels and neural networks (the “neural tangent kernel”) is an active research area that extends benign overfitting theory to non-linear models.

55 Historical Context

Section Summary

Benign overfitting challenges the statistical wisdom developed over a century. Understanding this history helps appreciate both why the classical view was dominant and why the modern revision is profound.

55.1 Classical Statistical Wisdom

The bias-variance tradeoff, formalised in the mid-20th century, became a cornerstone of statistical learning:

- **1960s–70s:** AIC, BIC, and cross-validation provide principled model selection
- **1980s–90s:** VC theory and PAC learning formalise complexity control
- **1990s–2000s:** SVMs, regularisation theory, kernel methods emphasise controlled complexity

The consistent message: match model complexity to data availability. Overparameterisation is dangerous.

55.2 The Modern Revolution

The deep learning revolution of the 2010s forced a reconsideration:

- **2012:** AlexNet wins ImageNet with 60 million parameters, far exceeding classical guidelines
- **2015–present:** Networks grow to billions of parameters; performance keeps improving
- **2018–present:** Theoretical work by Belkin, Bartlett, Ma, and others develops the double descent and benign overfitting framework

Key Theoretical Contributions

- **Belkin et al. (2019):** “Reconciling modern machine learning practice and the bias-variance trade-off”—introduced double descent for kernel methods and neural networks
- **Bartlett et al. (2020):** “Benign overfitting in linear regression”—provided precise conditions for when minimum-norm interpolation succeeds
- **Hastie et al. (2022):** “Surprises in high-dimensional ridgeless least squares interpolation”—detailed analysis of the interpolation threshold

These papers established that interpolation can be benign under specific, characterisable conditions—not randomly or magically, but for precise mathematical reasons.

55.3 Reconciling Old and New

The resolution isn't that classical theory was wrong—it was *incomplete*:

1. Classical theory focused on regimes where p and n are comparable
2. It correctly predicted disaster at the interpolation threshold $p \approx n$
3. But it didn't explore the $p \gg n$ regime where new phenomena emerge
4. The minimum-norm solution's implicit regularisation wasn't appreciated

Unified View

Classical and modern perspectives are both correct in their respective regimes:

Regime	Classical Prediction	Actual Behaviour
$p \ll n$	U-curve, optimal p exists	Correct
$p \approx n$	Error spikes	Correct
$p \gg n$	Catastrophic overfitting	Wrong —benign overfitting possible

The classical view was extrapolating from the $p \leq n$ regime into territory it hadn't mapped.

56 Practical Implications

Key Takeaways for Practice

1. **Interpolation isn't always bad:** In high dimensions with structured data, fitting training data perfectly can still generalise well
2. **Data structure matters:** The manifold hypothesis explains why—real data has low intrinsic dimension. Exploit this.
3. **Noise distribution matters:** High-dimensional noise must be spread across many directions (isotropic) for benign overfitting to work. Beware of systematic noise patterns.
4. **Implicit regularisation is powerful:** Minimum-norm solutions (from gradient descent or pseudoinverse) automatically avoid amplifying noise
5. **Don't fear overparameterisation:** Modern deep learning operates in the overparameterised regime successfully—with appropriate training procedures
6. **The interpolation threshold is dangerous:** If you must operate near $p \approx n$, use explicit regularisation (Ridge, dropout, early stopping)
7. **More data always helps for signal:** Even in high dimensions, more data improves estimation of the intrinsic structure (the k^*/n term)

NB!

Benign overfitting is **not** a license to ignore model complexity entirely. It requires:

- Data with genuine low-dimensional structure (the manifold hypothesis must hold)
- High-dimensional noise that is approximately isotropic
- Appropriate inductive biases (minimum norm, early stopping, architectural choices)
- Sufficient overparameterisation ($p \gg n$, not just $p > n$)

Without these conditions, overfitting remains harmful. The classical wisdom still applies when these conditions fail.

Standard regularisation (Ridge, Lasso, early stopping) remains important when:

- Data structure is unknown or weak
- Noise has systematic patterns
- You are near the interpolation threshold
- Computational constraints prevent reaching the overparameterised regime

57 Summary

Key Concepts from Week 5

1. **The overfitting paradox:** Classical theory predicts interpolating models should fail; modern practice shows they can succeed
2. **Regime-dependent behaviour:** OLS behaves fundamentally differently in low-dimensional ($p \ll n$) versus high-dimensional ($p \gg n$) settings
 - Low-dim: variance dominates, risk $\propto p/n$, more data helps rapidly
 - High-dim: bias dominates, risk $\propto (1 - n/p)\|\beta^*\|^2$, more data helps marginally
3. **Double descent:** Test error exhibits three regimes—classical U-curve, spike at interpolation threshold ($p \approx n$), and second descent in overparameterised regime ($p \gg n$)
4. **Interpolation threshold:** At $p \approx n$, the unique interpolating solution is maximally sensitive to noise; test error peaks
5. **Minimum-norm interpolation:** Among infinitely many interpolating solutions (when $p > n$), the minimum-norm solution generalises best and provides implicit regularisation
6. **SVD perspective:** Singular value decomposition separates signal (large singular values) from noise (small singular values); minimum-norm solutions automatically downweight noisy directions
7. **The k -split:** Decomposing β^* into signal and noise components reveals how the minimum-norm solution handles each part differently
8. **Conditions for benign overfitting:** Low intrinsic signal dimension ($k^* \ll p$), isotropic noise in high dimensions, and high effective rank in the noise subspace ($R_{k^*}(\Sigma) \gg n$)
9. **Manifold hypothesis:** Real data lies near low-dimensional manifolds, providing the structure that benign overfitting requires
10. **Connection to deep learning:** Neural networks benefit from overparameterisation through implicit regularisation in gradient descent, but the full picture involves feature learning beyond the linear theory
11. **Historical context:** Benign overfitting resolves the apparent contradiction between classical learning theory and modern deep learning practice

Connections to Other Weeks

- **Week 3:** Ridge and Lasso provide explicit regularisation; benign overfitting provides implicit regularisation. The connection $\lim_{\lambda \rightarrow 0^+} \hat{\beta}_{\text{ridge}} = \hat{\beta}_{\text{min-norm}}$ shows how these perspectives unify.
- **Week 4:** Classical generalisation bounds (VC dimension, Rademacher complexity) focus on $p < n$; benign overfitting extends the theory to $p > n$ where different phenomena emerge.
- **Week 10–11:** Neural networks exhibit benign overfitting; understanding the linear case illuminates deep learning. The implicit bias of gradient descent connects linear theory to deep networks.

58 Introduction to Kernel Methods

Chapter Overview

This chapter develops **kernel methods**—a family of algorithms that enable nonlinear learning through the elegant “kernel trick.” We cover:

- The dual view of regression: weighting observations by similarity
- Feature maps and the kernel trick: implicit high-dimensional computation
- Common kernels and their properties
- Kernel ridge regression and the representer theorem
- Support vector machines for classification
- Reproducing kernel Hilbert spaces (intuition)
- Practical considerations and limitations

Key insight: We can perform nonlinear learning while retaining the computational benefits of linear methods by working implicitly in high-dimensional feature spaces.

58.1 A New Way to Think About Learning

Kernel methods represent a fundamental shift in how we think about machine learning. Throughout this course, we have primarily asked: “Which features matter?” We assign coefficients to features and combine them linearly. But there is another equally valid question: “Which training examples are similar to my test point?”

This reframing—from *feature-centric* to *observation-centric*—leads to remarkably flexible models that can capture complex nonlinear relationships while remaining computationally tractable. The key insight is that many machine learning algorithms depend on the data only through *inner products* (dot products) between data points. If we can compute these inner products efficiently in a transformed space, we gain the benefits of that transformation without paying its computational cost.

Core Insight

Traditional regression asks: “Which features matter?”

Kernel methods ask: “Which training examples are similar to my test point?”

This reframing enables us to work with infinite-dimensional feature spaces tractably.

58.2 The Problem: Linear Methods Meet Nonlinear Data

Linear models are computationally efficient and well-understood, but real-world relationships are rarely linear. Consider classifying points in \mathbb{R}^2 where the decision boundary is circular—no hyperplane can separate the classes.

The traditional solution is **feature expansion**: transform inputs x via a feature map $\phi(x)$ into a higher-dimensional space where linear separation becomes possible. But this creates a new problem: if $\phi(x)$ is high-dimensional (or infinite-dimensional), explicit computation becomes intractable.

NB!

The feature expansion dilemma:

- **Low-dimensional features:** Computationally cheap, but may not capture nonlinear structure
- **High-dimensional features:** Can capture complex patterns, but computationally expensive

Kernel methods resolve this tension elegantly.

59 An Alternative View of Regression

Traditional regression assigns coefficients to *features*. An alternative perspective assigns weights to *observations* based on their similarity to the test point. This dual view is the gateway to kernel methods.

59.1 Two Equivalent Formulations of Ridge Regression

Consider ridge regression with the familiar solution $\hat{\beta} = (X^\top X + \lambda I_p)^{-1} X^\top y$. We can derive two equivalent prediction formulas using the matrix identity:

$$(A + \lambda I)^{-1} A = A(A + \lambda I)^{-1}$$

This identity—sometimes called the “push-through” identity—allows us to switch between inverting different matrices.

Two Views of Ridge Regression

Starting from the ridge regression solution $\hat{\beta} = (X^\top X + \lambda I_p)^{-1} X^\top y$, we can derive two equivalent prediction formulas:

Feature-space view (primal):

$$\hat{y} = X \hat{\beta}_{\text{ridge}} = X \underbrace{(X^\top X + \lambda I_p)^{-1}}_{p \times p} X^\top y$$

Observation-space view (dual):

$$\hat{y} = \underbrace{X X^\top}_{n \times n} (X X^\top + \lambda I_n)^{-1} y$$

Proof of equivalence: Let $A = X^\top$ and apply the identity with λI added appropriately. Alternatively, use the Woodbury matrix identity.

Let us unpack what these two formulations mean:

Feature-space view: We compute a $p \times p$ matrix $X^\top X$, which measures how features relate to each other across all observations. Each entry $(X^\top X)_{jk} = \sum_{i=1}^n x_{ij} x_{ik}$ is the dot product between feature columns j and k . We then invert this matrix and multiply by the design matrix and response.

Observation-space view: We compute an $n \times n$ matrix XX^\top , which measures how observations relate to each other across all features. Each entry $(XX^\top)_{ij} = \sum_{k=1}^p x_{ik}x_{jk} = x_i^\top x_j$ is the dot product between observation rows i and j . This captures **similarity between observations**.

When to Use Each View

- **Feature-space** ($p \times p$ matrix): Use when $p \ll n$ (few features, many observations)
- **Observation-space** ($n \times n$ matrix): Use when $p \gg n$ (many features, few observations)

The key insight: $X^\top X$ measures similarity between **features**, while XX^\top measures similarity between **observations**.

After feature expansion, p can become very large (even infinite), making the observation-space view essential.

The observation-space view is particularly powerful because it depends only on pairwise similarities between data points. This is the foundation of kernel methods: we will replace these dot products with more general similarity measures.

59.2 Similarity as Dot Product

The observation-space view reveals that predictions depend on similarities between data points. The dot product provides a natural measure of similarity, but why? Understanding this connection is crucial for kernel methods.

Dot Product and Distance

The squared Euclidean distance can be written in terms of dot products:

$$\begin{aligned} d^2(x, x') &= \|x - x'\|^2 = \sum_{i=1}^d (x_i - x'_i)^2 \\ &= \sum_{i=1}^d x_i^2 - 2 \sum_{i=1}^d x_i x'_i + \sum_{i=1}^d x'^2_i \\ &= x^\top x - 2x^\top x' + x'^\top x' \end{aligned}$$

For normalised vectors ($\|x\| = \|x'\| = 1$):

$$d^2(x, x') = 2(1 - x^\top x')$$

Thus **dot product \propto similarity \propto 1 - distance**.

This derivation shows that for normalised vectors, the dot product directly measures similarity: as the dot product increases, the distance decreases. The relationship $d^2 = 2(1 - x^\top x')$ makes this explicit—maximising the dot product minimises the distance.

The dot product has a beautiful geometric interpretation through the angle between vectors:

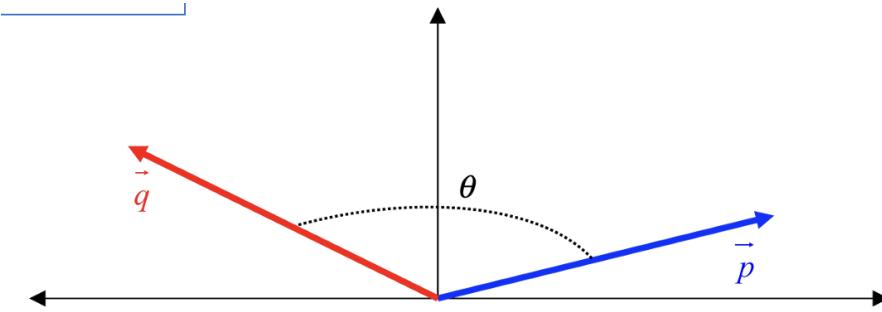


Figure 36: The dot product $x \cdot y = \|x\|\|y\| \cos \theta$ captures both magnitude and direction. When θ is small (similar directions), $\cos \theta$ is large and the dot product is large.

Dot Product as Similarity Measure

$$x \cdot y = \langle x, y \rangle = x^\top y = \|x\|\|y\| \cos \theta$$

This formula shows the dot product depends on:

1. **Magnitudes** ($\|x\|$ and $\|y\|$): Longer vectors produce larger dot products
2. **Direction** ($\cos \theta$): Vectors pointing similarly produce larger dot products

The directional component $\cos \theta$ is called **cosine similarity**:

$$\text{cosine similarity} = \frac{x \cdot y}{\|x\|\|y\|} = \cos \theta$$

This ranges from -1 (opposite directions) through 0 (orthogonal) to $+1$ (same direction).

Dot Product Interpretation

$$x \cdot y = \|x\|\|y\| \cos \theta$$

- $\theta = 0^\circ$: Parallel vectors \Rightarrow maximum similarity
- $\theta = 90^\circ$: Orthogonal vectors \Rightarrow no similarity
- $\theta = 180^\circ$: Anti-parallel vectors \Rightarrow maximum dissimilarity

Key intuition: If similar, angle is small \rightarrow cosine is large \rightarrow dot product is large.

Cosine similarity normalises by magnitude: $\cos \theta = \frac{x \cdot y}{\|x\|\|y\|}$

59.3 Regression as Weighted Averaging

In the observation-space view, prediction becomes a weighted average of training labels:

Prediction as Similarity-Weighted Average

For a test point \tilde{x} :

$$\hat{y}(\tilde{x}) = \sum_{i=1}^n w_i y_i$$

where the weights are:

$$w = \tilde{x}^\top X (X X^\top + \lambda I_n)^{-1}$$

Each weight w_i reflects how similar training point x_i is to the test point \tilde{x} .

Let us unpack this formula. The term $\tilde{x}^\top X$ computes the dot product between the test point \tilde{x} and each column of X^\top —that is, each training observation. These raw similarities are then transformed by the matrix $(X X^\top + \lambda I_n)^{-1}$, which accounts for the relationships among training points and the regularisation.

The result is intuitive: observations more similar to \tilde{x} receive higher weights in the prediction. We are “taking a walk” through the training data, finding observations similar to our test point, and using their labels to make predictions. The regularisation parameter λ prevents overfitting by smoothing the weights.

NB!

In linear regression, weights evolve **linearly** with the dot product $\tilde{x}^\top x_i$. This is inflexible—we may want nearby points to have much higher weight than distant ones, or we may want similarity to depend on nonlinear relationships.

Consider predicting at $\tilde{x} = 2$: with linear weighting, a training point at $x = 3$ might receive more weight than one at $x = 2.01$, simply due to the global structure of the linear model.

This motivates **nonlinear similarity measures**: kernels.

59.4 The Importance of Defining Similarity Correctly

Different notions of similarity lead to fundamentally different models. Our choice of similarity measure determines the kinds of functions we can learn—and the wrong choice can be disastrous.

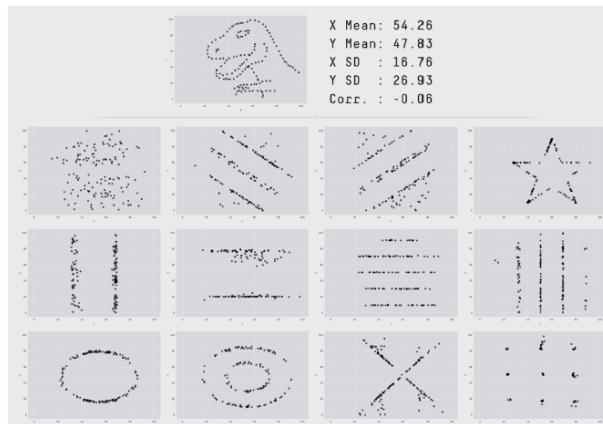


Figure 37: All four datasets have the same correlation coefficient, but their structures differ dramatically. Linear correlation (based on Euclidean distance) cannot distinguish them. This is Anscombe’s quartet.

NB!

Critical insight: Our measure of similarity determines the kinds of functions we can learn.

If we only measure linear correlation based on Euclidean distance, we will estimate the same covariance structure for all four datasets above. Different problems demand different notions of distance and similarity.

If you define distance differently, you get different measures of similarity. Two points that are “close” in Euclidean distance may not be “close” in a transformed feature space—and this flexibility is precisely what makes kernel methods powerful.

60 Feature Maps and the Kernel Trick

Section Summary: The Kernel Trick

The problem: We want to work in high-dimensional feature spaces for expressive power, but computing $\phi(x)$ explicitly is expensive or impossible.

The solution: Many algorithms depend on data only through dot products $\langle x_i, x_j \rangle$. If we can compute $k(x, x') = \langle \phi(x), \phi(x') \rangle$ directly without computing ϕ , we get the benefits of high-dimensional features at low cost.

The insight: A **kernel function** $k(x, x')$ computes inner products in feature space implicitly.

60.1 Feature Expansion

To capture nonlinear relationships with linear models, we transform the input space. This is the idea of *feature expansion* that we have used throughout this course—polynomial regression and Fourier basis expansions are examples.

Feature Map

A **feature map** $\phi : \mathcal{X} \rightarrow \mathcal{H}$ transforms inputs from the original space \mathcal{X} into a (typically higher-dimensional) feature space \mathcal{H} .

Examples:

- **Polynomial:** $\phi(x) = [1, x, x^2, \dots, x^M]^\top$ for $x \in \mathbb{R}$
- **Trigonometric:** $\phi(x) = [1, \cos(x), \sin(x), \cos(2x), \sin(2x), \dots]^\top$
- **Interaction terms:** $\phi([x_1, x_2]^\top) = [x_1, x_2, x_1 x_2, x_1^2, x_2^2]^\top$

In the expanded space, linear models can capture nonlinear relationships in the original space.

Why this works: A function that is nonlinear in x may be linear in $\phi(x)$. For example, $y = ax^2 + bx + c$ is nonlinear in x but linear in $\phi(x) = [1, x, x^2]^\top$ with coefficients $[c, b, a]^\top$. The key insight is that linear regression in a transformed feature space is equivalent to nonlinear regression in the original space.

Concrete example: For a 2D input $x = [x_1, x_2]^\top$, a polynomial expansion might be:

$$\phi(x) = [x_1, x_2, x_1^2, x_2^2, x_1 x_2]^\top$$

This maps 2 features to 5 features, enabling the model to capture quadratic relationships.

The problem: As feature dimension grows, computation becomes expensive:

- Polynomial features of degree M in d dimensions: $\binom{d+M}{M}$ features
- For $d = 100$, $M = 5$: over 96 million features
- Some kernels correspond to *infinite*-dimensional feature spaces

Computing $\phi(x)^\top \phi(x')$ explicitly becomes intractable when ϕ is very high- or infinite-dimensional.

60.2 The Kernel Trick

The kernel trick is one of the most elegant ideas in machine learning. It allows us to work in high-dimensional (even infinite-dimensional) feature spaces without ever explicitly computing the features.

Kernel Definition

A **kernel** is a function $k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ that computes the inner product in some feature space:

$$k(x, x') = \langle \phi(x), \phi(x') \rangle_{\mathcal{H}}$$

for some feature map $\phi : \mathcal{X} \rightarrow \mathcal{H}$.

The **kernel trick**: In algorithms that depend on data only through inner products $\langle x_i, x_j \rangle$, replace these with kernel evaluations $k(x_i, x_j)$ to implicitly work in the feature space \mathcal{H} .

The beauty of kernel methods is their ability to *implicitly* compute dot products in high-dimensional feature spaces without ever explicitly constructing the feature vectors $\phi(x)$ and $\phi(x')$.

To summarise the key objects:

- $\phi(x)$ is a **feature expansion**: a function that transforms input x into a higher-dimensional space
- $k(x, x') = \phi(x)^\top \phi(x')$ is a **kernel function**: computes similarity in the expanded space
- The kernel is a **similarity metric**—it measures how similar two inputs are in the transformed feature space

Let us see this in action with a concrete example.

Worked Example: Polynomial Kernel

Consider $x, z \in \mathbb{R}^2$ and the kernel $k(x, z) = (x^\top z)^2$.

Step 1: Expand the kernel algebraically.

$$\begin{aligned} k(x, z) &= (x^\top z)^2 = (x_1 z_1 + x_2 z_2)^2 \\ &= x_1^2 z_1^2 + 2x_1 x_2 z_1 z_2 + x_2^2 z_2^2 \end{aligned}$$

Step 2: Identify this as an inner product.

$$k(x, z) = \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1 x_2 \\ x_2^2 \end{bmatrix}^\top \begin{bmatrix} z_1^2 \\ \sqrt{2}z_1 z_2 \\ z_2^2 \end{bmatrix} = \phi(x)^\top \phi(z)$$

where $\phi(x) = (x_1^2, \sqrt{2}x_1 x_2, x_2^2)^\top$.

Computational comparison:

- Direct kernel evaluation: 2 multiplications + 1 addition + 1 squaring = $O(d)$
- Explicit feature computation: Compute $\phi(x), \phi(z)$, then inner product = $O(d^2)$

The kernel computes the same result more efficiently, and this gap widens dramatically for higher degrees.

Notice the $\sqrt{2}$ factor in the feature map. This is necessary to make the cross-term $2x_1 x_2 z_1 z_2$ emerge from the dot product. The kernel “knows” about this scaling implicitly—we never need to compute it.

General Polynomial Kernel

For the kernel $k(x, z) = (x^\top z + c)^M$ with $x, z \in \mathbb{R}^d$:

- The implicit feature space has dimension $\binom{d+M}{M}$
- Direct kernel evaluation: $O(d)$ operations
- Explicit feature computation: $O\left(\binom{d+M}{M}\right)$ operations

For $d = 100, M = 3$: kernel takes $O(100)$ operations; explicit features would take $O(176,851)$ operations.

For $d = 100, M = 10$: kernel takes $O(100)$ operations; explicit features would require computing in a space of dimension $\binom{110}{10} \approx 10^{13}!$

60.3 The Gram Matrix

When applying kernel methods to a dataset, we need to compute all pairwise kernel evaluations. This information is collected in the *Gram matrix*.

Gram Matrix

The **Gram matrix** (or kernel matrix) $K \in \mathbb{R}^{n \times n}$ contains all pairwise kernel evaluations:

$$K_{ij} = k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$$

In terms of the feature matrix $\Phi = [\phi(x_1), \dots, \phi(x_n)]^\top$:

$$K = \Phi \Phi^\top$$

Properties:

- K is symmetric: $K_{ij} = K_{ji}$
- K is positive semi-definite: $\alpha^\top K \alpha \geq 0$ for all $\alpha \in \mathbb{R}^n$

The Gram matrix replaces XX^\top in kernelised algorithms. We store K (which is $n \times n$) rather than Φ (which could be $n \times \infty$). This is the key computational insight: regardless of how high-dimensional the implicit feature space is, we only ever work with the $n \times n$ Gram matrix.

Interpretation: Entry K_{ij} measures the similarity between observations i and j in the feature space. A valid kernel produces a positive semi-definite Gram matrix, meaning all eigenvalues are ≥ 0 . This ensures the kernel “looks like” a dot product—it must arise from some (possibly implicit) feature map.

61 What Makes a Valid Kernel?

Not every function $k(x, x')$ corresponds to an inner product in some feature space. Mercer’s theorem characterises valid kernels.

Section Summary: Valid Kernels

A function $k(x, x')$ is a valid kernel if and only if:

1. It is **symmetric**: $k(x, x') = k(x', x)$
2. It is **positive semi-definite**: The Gram matrix K has all eigenvalues ≥ 0 for any finite set of points

This ensures k corresponds to an inner product in *some* Hilbert space.

Mercer's Theorem (Informal)

Let $k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ be a continuous symmetric function. Then k is a valid kernel (i.e., corresponds to an inner product in some Hilbert space) if and only if it is **positive semi-definite**:

For any finite set $\{x_1, \dots, x_n\} \subset \mathcal{X}$ and any $\alpha \in \mathbb{R}^n$:

$$\sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j k(x_i, x_j) \geq 0$$

Equivalently, the Gram matrix $K_{ij} = k(x_i, x_j)$ is positive semi-definite for all finite point sets.

Interpretation: PSD ensures that “distances” computed via k behave sensibly—you cannot have negative squared distances in the implicit feature space.

The condition $\sum_{i,j} \alpha_i \alpha_j k(x_i, x_j) \geq 0$ can be written in matrix form as $\alpha^\top K \alpha \geq 0$. This is precisely the definition of a positive semi-definite matrix. Intuitively, this ensures that the “geometry” induced by the kernel is consistent—distances are non-negative, the triangle inequality holds, and so forth.

Why does this matter? If we try to use a function that is not a valid kernel, we lose the guarantees that make kernel methods work. The optimisation problems may become non-convex, solutions may not exist, and the interpretation as working in a feature space breaks down.

61.1 Constructing Kernels

One of the most powerful aspects of kernel methods is that kernels can be combined to express complex notions of similarity. The following closure properties allow us to build sophisticated kernels from simpler building blocks.

Closure Properties of Kernels

If k_1 and k_2 are valid kernels, then so are:

1. αk_1 for any $\alpha \geq 0$ (scaling)
2. $k_1 + k_2$ (sum)
3. $k_1 \cdot k_2$ (product)
4. $f(x)k_1(x, x')f(x')$ for any function $f : \mathcal{X} \rightarrow \mathbb{R}$
5. $g(k_1(x, x'))$ where g is a polynomial with non-negative coefficients
6. $\exp(k_1(x, x'))$ (exponential)

These rules let us build complex kernels from simpler building blocks.

Let us understand why these rules work:

Sum of kernels: If k_1 corresponds to feature map ϕ_1 and k_2 to ϕ_2 , then $k_1 + k_2$ corresponds to the concatenated feature map $[\phi_1; \phi_2]$. The resulting feature space is the direct sum of the individual spaces.

Product of kernels: The product $k_1 \cdot k_2$ corresponds to a feature space containing all pairwise

products of features from ϕ_1 and ϕ_2 . This creates an even higher-dimensional space capturing interactions.

Exponential of kernels: This follows from the Taylor expansion $e^x = \sum_{n=0}^{\infty} x^n/n!$ and the fact that powers of valid kernels are valid (via the product rule).

Manipulating Kernels

allows you to construct complicated kernels

- Given two kernels $k_1(x, x')$ and $k_2(x, x')$, the following are all valid:

- | | |
|--|--|
| <ul style="list-style-type: none"> • $ck_1(x, x')$ <ul style="list-style-type: none"> • Multiply by a constant • $f(x)k_1(x, x')f(x')$ <ul style="list-style-type: none"> • Pre and multiply by some function of the inputs • $\text{poly}(k_1(x, x'))$ <ul style="list-style-type: none"> • Pass it through a polynomial • $\exp(k_1(x, x'))$ <ul style="list-style-type: none"> • Pass it through an exponential | <ul style="list-style-type: none"> • $k_1(x, x') + k_2(x, x')$ <ul style="list-style-type: none"> • Add two together • $k_1(x, x')k_2(x, x')$ <ul style="list-style-type: none"> • Multiply them together • $k_3(\phi(x), \phi(x'))$ <ul style="list-style-type: none"> • Expand feature and pass through a valid kernel • $x^T Ax'$ <ul style="list-style-type: none"> • A bilinear form with the symmetric PSD matrix A |
|--|--|

|

Figure 38: Rules for constructing valid kernels from simpler ones. Any combination that preserves positive semi-definiteness yields a valid kernel.

Kernel Combinations

Custom kernels can encode domain knowledge about similarity:

$$k_{\text{custom}} = \alpha \cdot k_{\text{local}} + (1 - \alpha) \cdot k_{\text{global}}$$

Example applications:

- **Strings:** Count common substrings
- **Graphs:** Compare random walk distributions
- **Images:** Combine spatial and colour similarity
- **Time series:** Combine trend and seasonal components

The hyperparameter α balances the contributions of different similarity notions.

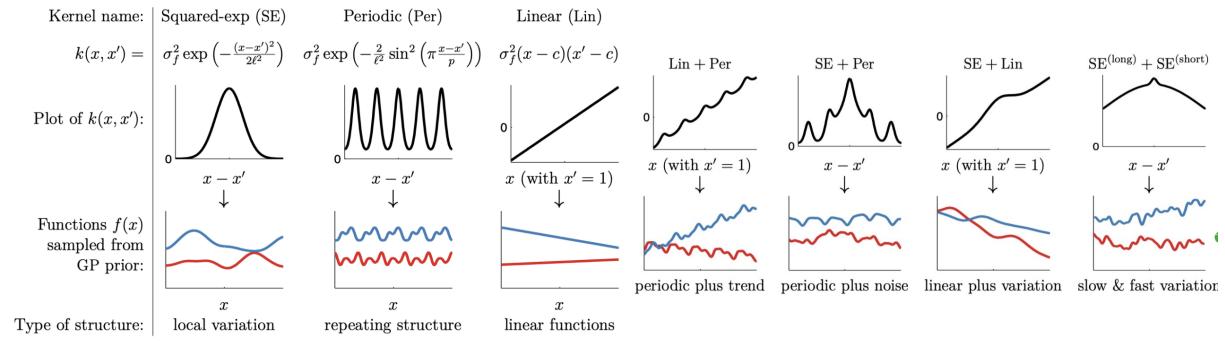


Figure 39: Combining kernels: Gaussian (local), periodic (cyclical), and mixed. Adjusting σ^2 combines wide global structure with high-frequency local variation.

The figure illustrates how different kernels capture different aspects of structure:

- **Gaussian kernel:** Captures locality—nearby points are similar
- **Periodic kernel:** Captures cyclical patterns—e.g., days of the week have similarity to each other
- **Combined:** A low-frequency wide global model combined with a high-frequency local model

This flexibility allows you to express prior beliefs about your data: perhaps there is a cyclical time trend, or one part of the feature space benefits from local models while another requires global structure.

62 Common Kernels

Section Summary: Common Kernels

Kernel	Formula	Feature Dimension
Linear	$k(x, z) = x^\top z$	d
Polynomial	$k(x, z) = (x^\top z + c)^M$	$\binom{d+M}{M}$
RBF/Gaussian	$k(x, z) = \exp\left(-\frac{\ x-z\ ^2}{2\sigma^2}\right)$	∞

Rule of thumb: Start with RBF for general problems; use polynomial when interactions up to a specific degree are meaningful; use linear as a baseline.

62.1 Linear Kernel

The simplest kernel is the linear kernel, which corresponds to no feature transformation at all.

Linear Kernel

$$k(x, z) = x^\top z$$

Properties:

- Feature map: $\phi(x) = x$ (identity)
- Feature dimension: d (same as input)
- Corresponds to standard linear methods

When to use:

- As a baseline before trying nonlinear kernels
- When $d \gg n$ (high-dimensional sparse data, e.g., text)
- When relationships are approximately linear

The linear kernel provides no additional expressiveness but serves as a useful baseline and is computationally cheapest.

Using the linear kernel in a kernel algorithm is equivalent to using the corresponding non-kernelised algorithm. For example, kernel ridge regression with a linear kernel is equivalent to standard ridge regression. This makes the linear kernel a useful sanity check: if a nonlinear kernel does not improve over linear, either the relationship is genuinely linear or the nonlinear kernel is poorly tuned.

62.2 Polynomial Kernel

The polynomial kernel implicitly computes dot products in a space containing all polynomial terms up to a specified degree.

Polynomial Kernel

$$k(x, z) = (x^\top z + c)^M$$

Parameters:

- M : Degree (higher = more complex decision boundaries)
- $c \geq 0$: Trade-off between lower and higher-order terms
 - $c = 0$: Homogeneous polynomial (only degree- M terms)
 - $c > 0$: Includes all terms from degree 0 to M

Feature space: Contains all monomials up to degree M :

$$\phi(x) \propto [\dots, x_{i_1}^{a_1} x_{i_2}^{a_2} \cdots x_{i_k}^{a_k}, \dots]$$

where $a_1 + a_2 + \cdots + a_k \leq M$.

When to use:

- When interaction effects between features matter
- When domain knowledge suggests polynomial relationships
- Image processing (e.g., polynomial SVMs for digit recognition)

The constant c controls which polynomial terms are emphasised. Let us see this with a detailed example.

Detailed Example: Polynomial Kernel with $c > 0$

Let $x, z \in \mathbb{R}^2$ and $k(x, z) = (x^\top z + 1)^2$.

Expanding:

$$\begin{aligned} k(x, z) &= (x_1 z_1 + x_2 z_2 + 1)^2 \\ &= x_1^2 z_1^2 + x_2^2 z_2^2 + 1 + 2x_1 x_2 z_1 z_2 + 2x_1 z_1 + 2x_2 z_2 \\ &= \phi(x)^\top \phi(z) \end{aligned}$$

where $\phi(x) = (x_1^2, x_2^2, 1, \sqrt{2}x_1 x_2, \sqrt{2}x_1, \sqrt{2}x_2)^\top$.

This feature space includes:

- Constant term: 1 (from $c = 1$)
- Linear terms: x_1, x_2
- Quadratic terms: $x_1^2, x_2^2, x_1 x_2$

Notice that with $c = 1$, the feature space includes terms of all degrees from 0 to 2. If we had used $c = 0$ (the homogeneous polynomial kernel), only the degree-2 terms would appear. The choice of c thus determines whether lower-order terms contribute to similarity.

62.3 Gaussian (RBF) Kernel

The Gaussian kernel, also called the Radial Basis Function (RBF) kernel, is perhaps the most widely used kernel. It corresponds to an infinite-dimensional feature space and can approximate any continuous function.

Gaussian/RBF Kernel

$$k(x, z) = \exp\left(-\frac{\|x - z\|^2}{2\sigma^2}\right) = \exp(-\gamma\|x - z\|^2)$$

where $\gamma = \frac{1}{2\sigma^2}$.

Properties:

- **Bounded:** $k(x, z) \in (0, 1]$, with $k(x, x) = 1$
- **Local:** Similarity decays exponentially with distance
- **Infinite-dimensional:** Corresponds to an infinite-dimensional feature space

Parameters:

- Small σ (large γ): Highly local; only very close points are similar
- Large σ (small γ): More global; distant points retain similarity

The formula has an intuitive interpretation:

1. Take the difference between x and z
2. Square the differences and sum them (squared Euclidean distance)
3. Normalise by $2\sigma^2$
4. Apply the exponential (which decays as distance increases)

The result is a similarity measure that equals 1 when $x = z$ and decays smoothly towards 0 as points become distant. The bandwidth parameter σ controls how quickly this decay happens.

62.3.1 Why the RBF Kernel is Infinite-Dimensional

The RBF kernel's power comes from its implicit infinite-dimensional feature space. We can see this by expanding the kernel using the Taylor series.

Why RBF is Infinite-Dimensional

Expanding the RBF kernel using Taylor series:

$$\begin{aligned} k(x, z) &= \exp\left(-\frac{\|x\|^2}{2\sigma^2}\right) \exp\left(\frac{x^\top z}{\sigma^2}\right) \exp\left(-\frac{\|z\|^2}{2\sigma^2}\right) \\ &= e^{-\|x\|^2/2\sigma^2} \cdot e^{-\|z\|^2/2\sigma^2} \cdot \sum_{k=0}^{\infty} \frac{(x^\top z)^k}{\sigma^{2k} k!} \end{aligned}$$

Each term $(x^\top z)^k$ corresponds to a polynomial kernel of degree k . Thus the RBF kernel implicitly uses an **infinite-dimensional** feature space containing all polynomial terms, with higher-degree terms down-weighted by $\frac{1}{\sigma^{2k} k!}$.

Intuition: The RBF kernel “prefers” smoother (lower-degree) functions through this weighting, while retaining the flexibility to capture arbitrary local variation when the data demands it.

The key insight from the Taylor expansion is that the RBF kernel contains *all* polynomial terms of all degrees, but with weights that decrease rapidly as the degree increases. The k -th component of the implicit feature map has the form:

$$\phi(x)_k \propto \exp\left(-\frac{\|x\|^2}{2\sigma^2}\right) \frac{x^k}{\sigma^k \sqrt{k!}}$$

Because $k!$ grows extremely rapidly, **higher-degree terms are weighted down exponentially**. This means the RBF kernel “prefers” smoother, lower-degree functions while retaining flexibility for local variation.

This is analogous to regularisation in Fourier series, where we down-weight high-frequency components (e.g., $\cos(mx)/m$ for large m). The Gaussian kernel achieves similar smoothness implicitly through its infinite-dimensional feature space.

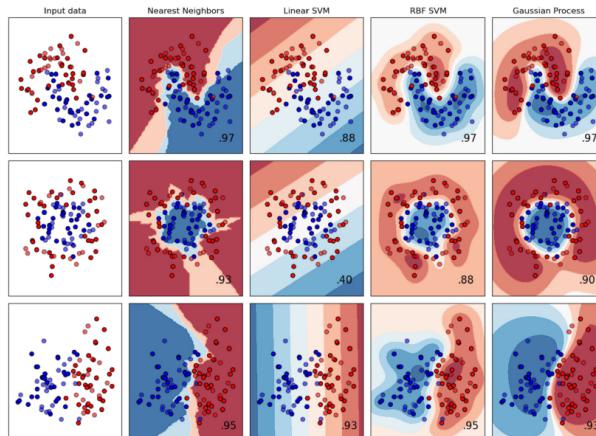


Figure 40: Left: Linear regression (rigid, global). Right: RBF kernel regression allows locally-weighted similarity, adapting flexibly to the data structure.

Choosing σ for RBF

The bandwidth σ controls the bias-variance trade-off:

- **Small σ :** High variance, low bias (can fit noise; decision boundaries follow individual points)
- **Large σ :** Low variance, high bias (overly smooth; approaches linear)

Heuristics:

- Median heuristic: Set σ to the median pairwise distance
- Cross-validation: Search over $\gamma \in \{10^{-3}, 10^{-2}, \dots, 10^3\}$

The bandwidth σ is perhaps the most important hyperparameter in RBF kernel methods. Too small, and the model overfits—each training point becomes an isolated “island” with no influence on neighbours. Too large, and the model underfits—all points are considered similar, reducing to a constant prediction.

62.4 Other Common Kernels

Beyond polynomial and RBF kernels, many other kernels exist for specific applications:

- **Periodic kernel:** $k(x, z) = \exp\left(-\frac{2\sin^2(\pi|x-z|/p)}{\sigma^2}\right)$ for cyclical/seasonal patterns
- **String kernels:** For text and sequence data, measuring similarity based on shared substrings
- **Graph kernels:** For structured/relational data, comparing random walk distributions or subgraph patterns
- **Matérn kernels:** A family generalising RBF with controllable differentiability

The choice of kernel encodes your beliefs about the structure of similarity in your problem. Domain expertise can guide this choice—for time series with known periodicity, a periodic kernel is natural; for molecular data, graph kernels respect molecular structure.

63 Kernel Ridge Regression

Section Summary: Kernel Ridge Regression

Kernel ridge regression performs ridge regression in the feature space defined by a kernel, without explicitly computing features:

$$\hat{y}(\tilde{x}) = k_{\tilde{x}}^\top (K + \lambda I)^{-1} y$$

Key insights:

- Solves an $n \times n$ system (not $D \times D$ where D may be infinite)
- The representer theorem guarantees the solution lies in the span of training kernel evaluations
- Computational cost: $O(n^3)$ for training, $O(n)$ for prediction

63.1 Derivation via the Dual

We now derive kernel ridge regression by starting from ridge regression in the (implicit) feature space and showing that we never need to compute features explicitly.

Primal Problem

Ridge regression in feature space minimises:

$$\min_w \frac{1}{2} \|y - \Phi w\|^2 + \frac{\lambda}{2} \|w\|^2$$

where $\Phi = [\phi(x_1), \dots, \phi(x_n)]^\top \in \mathbb{R}^{n \times D}$ is the feature matrix.

The primal solution is:

$$w^* = (\Phi^\top \Phi + \lambda I_D)^{-1} \Phi^\top y$$

Problem: If D is large or infinite, this is intractable.

This is precisely the situation where the kernel trick saves us. The key insight comes from the representer theorem, which tells us that even though w lives in a potentially infinite-dimensional space, we can express it using only n coefficients.

The Representer Theorem

Theorem: For any regularised empirical risk minimisation problem of the form:

$$\min_w \sum_{i=1}^n L(y_i, \langle w, \phi(x_i) \rangle) + \Omega(\|w\|)$$

where L is any loss function and Ω is a strictly increasing function, the optimal solution has the form:

$$w^* = \sum_{i=1}^n \alpha_i \phi(x_i) = \Phi^\top \alpha$$

for some $\alpha \in \mathbb{R}^n$.

Interpretation: Even though w lives in a potentially infinite-dimensional space, it can be written as a linear combination of the (finite) training feature vectors. We need only find the n coefficients α_i .

The representer theorem is one of the most important results in kernel methods. It says that no matter how high-dimensional the feature space is, the optimal solution can always be written as a linear combination of the training feature vectors. This reduces an infinite-dimensional optimisation problem to a finite-dimensional one.

Intuition: Why should w^* lie in the span of $\{\phi(x_1), \dots, \phi(x_n)\}$? The regulariser $\|w\|^2$ penalises any component of w that is orthogonal to this span. Since such components do not affect predictions on training data (they are orthogonal to all $\phi(x_i)$), they only add to the penalty without reducing the loss. Thus the optimal w^* has no such components.

Dual Derivation

Step 1: Apply the representer theorem. Since $w^* = \Phi^\top \alpha$:

- Predictions: $\Phi w^* = \Phi \Phi^\top \alpha = K\alpha$
- Regulariser: $\|w^*\|^2 = \alpha^\top \Phi \Phi^\top \alpha = \alpha^\top K\alpha$

Step 2: Rewrite the objective in terms of α :

$$\min_{\alpha} \frac{1}{2} \|y - K\alpha\|^2 + \frac{\lambda}{2} \alpha^\top K\alpha$$

Step 3: Take derivative and set to zero:

$$\begin{aligned} \frac{\partial}{\partial \alpha} &= -K(y - K\alpha) + \lambda K\alpha = 0 \\ &\Rightarrow K(K + \lambda I)\alpha = Ky \\ &\Rightarrow \alpha^* = (K + \lambda I)^{-1}y \end{aligned}$$

Step 4: Prediction for new point \tilde{x} :

$$\hat{y}(\tilde{x}) = \langle w^*, \phi(\tilde{x}) \rangle = \sum_{i=1}^n \alpha_i^* k(x_i, \tilde{x}) = k_{\tilde{x}}^\top (K + \lambda I)^{-1} y$$

where $k_{\tilde{x}} = [k(\tilde{x}, x_1), \dots, k(\tilde{x}, x_n)]^\top$.

Let us trace through the key steps:

Step 1: Using $w^* = \Phi^\top \alpha$, the predictions become $\Phi w^* = \Phi \Phi^\top \alpha = K\alpha$, where $K = \Phi \Phi^\top$ is the Gram matrix. Similarly, $\|w^*\|^2 = \alpha^\top \Phi \Phi^\top \alpha = \alpha^\top K\alpha$. Notice that everything now depends on Φ only through K .

Step 2: Substituting into the ridge regression objective gives us an optimisation problem in α that involves only the kernel matrix K .

Step 3: Taking derivatives and solving, we get $\alpha^* = (K + \lambda I)^{-1}y$. This involves inverting an $n \times n$ matrix, which is tractable even when the feature space is infinite-dimensional.

Step 4: For prediction, we compute kernel evaluations between the test point and all training points, then take a weighted combination.

Kernel Ridge Regression: Final Form

Training: Compute the Gram matrix $K_{ij} = k(x_i, x_j)$ and solve:

$$\alpha^* = (K + \lambda I_n)^{-1} y$$

Prediction: For new point \tilde{x} :

$$\hat{y}(\tilde{x}) = \sum_{i=1}^n \alpha_i^* k(\tilde{x}, x_i) = k_{\tilde{x}}^\top \alpha^*$$

Complexity:

- Gram matrix: $O(n^2)$ kernel evaluations
- Matrix inversion: $O(n^3)$
- Prediction: $O(n)$ per test point

Properties of Kernel Ridge Regression:

- **Global:** All training points contribute to each prediction (though distant points may contribute little with localised kernels)
- **Flexible:** Choice of kernel determines the notion of similarity
- **Closed-form:** Unlike many kernel methods, KRR has an analytical solution

Hyperparameters to tune:

- Choice of kernel (and its parameters, e.g., σ for RBF)
- Regularisation parameter λ

Computational Trade-off

	Primal (explicit features)	Dual (kernel)
Training	$O(nd^2 + d^3)$	$O(n^2 d_k + n^3)$
Prediction	$O(d)$	$O(n)$

Where d_k is the cost of one kernel evaluation.

Use primal when: d is moderate and n is large.

Use dual when: d is very large (or infinite) and n is moderate.

The prediction cost deserves attention: with kernel methods, prediction requires $O(n)$ kernel evaluations (one for each training point), whereas with explicit features, prediction is $O(d)$. This means kernel methods become expensive for prediction when n is large, even though training remains tractable.

64 Support Vector Machines

Section Summary: Support Vector Machines

SVMs are maximum-margin classifiers that:

- Find the hyperplane that separates classes with the **largest margin**
- Use only a subset of training points (**support vectors**) for prediction
- Handle non-separable data via **soft margins** and **slack variables**
- Achieve nonlinear decision boundaries via the **kernel trick**

Key formula (dual): $\alpha^* = \arg \max_{\alpha} \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j k(x_i, x_j)$ subject to constraints.

Support Vector Machines (SVMs) were one of the most successful machine learning algorithms before the deep learning era, and remain important today. They combine the kernel trick with a clever objective function that leads to sparse solutions.

64.1 Maximum Margin Classification

Consider binary classification with labels $y_i \in \{-1, +1\}$. A linear classifier predicts:

$$\hat{y}(x) = \text{sign}(w^\top x + b)$$

If the data is linearly separable, infinitely many hyperplanes achieve zero training error. Which should we choose?

The SVM answer is: choose the hyperplane that maximises the *margin*—the distance from the hyperplane to the nearest training point.

Geometric Margin

For a hyperplane $w^\top x + b = 0$, the **geometric margin** of a point (x_i, y_i) is:

$$\gamma_i = y_i \cdot \frac{w^\top x_i + b}{\|w\|}$$

This is the signed distance from x_i to the hyperplane, positive if correctly classified.

The **margin** of the classifier is:

$$\gamma = \min_i \gamma_i$$

the distance from the hyperplane to the nearest point.

Let us unpack this formula. The quantity $(w^\top x_i + b)/\|w\|$ is the signed distance from point x_i to the hyperplane (this follows from the geometry of hyperplanes). Multiplying by y_i makes this positive when the point is correctly classified. The margin $\gamma = \min_i \gamma_i$ is thus the distance to the nearest point, which is also the point most likely to be misclassified under small perturbations.

Why Maximise Margin?

Geometric intuition: A larger margin means the classifier is more “confident”—small perturbations to test points are less likely to change predictions.

Statistical intuition: Margin is inversely related to VC dimension. Larger margin \Rightarrow lower complexity \Rightarrow better generalisation bounds.

Robustness: Maximum margin classifiers are optimal under certain noise models (e.g., bounded perturbations).

The statistical argument is particularly important. The margin controls the “effective complexity” of the classifier: a large-margin classifier uses less of its capacity to fit the training data, leaving more capacity for generalisation. This connects to the bias-variance trade-off—larger margin means more bias (simpler decision boundary) but less variance (more stable under perturbations).

64.2 Hard-Margin SVM

The hard-margin SVM assumes the data is linearly separable and finds the maximum-margin hyperplane.

Hard-Margin SVM: Primal Formulation

Objective: Maximise the margin while correctly classifying all points.

Since $\gamma = 1/\|w\|$ when we normalise so that $\min_i |w^\top x_i + b| = 1$, maximising margin is equivalent to minimising $\|w\|$:

$$\min_{w,b} \frac{1}{2} \|w\|^2 \quad \text{subject to} \quad y_i(w^\top x_i + b) \geq 1 \quad \forall i$$

Constraints: Each training point must be on the correct side of the margin boundary.

The normalisation $\min_i |w^\top x_i + b| = 1$ fixes the scale of w (since we can always rescale w and b together). With this normalisation, the margin is exactly $1/\|w\|$, so maximising margin is equivalent to minimising $\|w\|^2$ (the square is for mathematical convenience—it makes the problem quadratic).

Hard-Margin SVM: Dual Formulation

Using Lagrange multipliers $\alpha_i \geq 0$ for each constraint:

Lagrangian:

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2} \|w\|^2 - \sum_{i=1}^n \alpha_i [y_i(w^\top x_i + b) - 1]$$

KKT conditions (setting derivatives to zero):

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w} &= 0 \Rightarrow w = \sum_{i=1}^n \alpha_i y_i x_i \\ \frac{\partial \mathcal{L}}{\partial b} &= 0 \Rightarrow \sum_{i=1}^n \alpha_i y_i = 0\end{aligned}$$

Dual problem (substituting back):

$$\max_{\alpha} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j (x_i^\top x_j)$$

$$\text{subject to } \alpha_i \geq 0, \quad \sum_i \alpha_i y_i = 0$$

Key observation: The objective depends on data only through inner products $x_i^\top x_j$!

The dual formulation is remarkable for two reasons:

1. **It depends only on inner products:** This means we can apply the kernel trick directly, replacing $x_i^\top x_j$ with $k(x_i, x_j)$.
2. **It leads to sparse solutions:** The KKT conditions imply that most $\alpha_i = 0$, so only a subset of training points (the support vectors) contribute to the solution.

Support Vectors

By the KKT complementary slackness conditions:

$$\alpha_i [y_i(w^\top x_i + b) - 1] = 0$$

Either $\alpha_i = 0$ (point is not used) or $y_i(w^\top x_i + b) = 1$ (point is exactly on the margin).

Support vectors: Points with $\alpha_i > 0$. These are the points lying exactly on the margin boundaries $w^\top x + b = \pm 1$.

Prediction depends only on support vectors:

$$\hat{y}(x) = \text{sign} \left(\sum_{i:\alpha_i>0} \alpha_i y_i (x_i^\top x) + b \right)$$

Sparsity: Typically only a small fraction of training points are support vectors, making prediction efficient.

The complementary slackness condition is the key to understanding support vectors. It says

that for each constraint, either the Lagrange multiplier is zero (constraint is not active) or the constraint is satisfied with equality (point is exactly on the margin). Points strictly inside the margin have $\alpha_i = 0$ and do not contribute to w .

This sparsity is a major advantage of SVMs over kernel ridge regression. In KRR, all training points contribute to predictions; in SVMs, only the support vectors matter. For large datasets, this can mean dramatic computational savings at prediction time.

64.3 Soft-Margin SVM

Real data is rarely linearly separable. Soft-margin SVMs allow some misclassification via **slack variables**.

Soft-Margin SVM: Primal Formulation

Introduce slack variables $\xi_i \geq 0$ to allow margin violations:

$$\begin{aligned} & \min_{w,b,\xi} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^n \xi_i \\ & \text{subject to } y_i(w^\top x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0 \end{aligned}$$

Interpretation:

- $\xi_i = 0$: Point correctly classified outside margin
- $0 < \xi_i < 1$: Point correctly classified but inside margin
- $\xi_i \geq 1$: Point misclassified

Parameter C : Trade-off between margin maximisation and error tolerance.

- Large C : Penalise errors heavily; small margin, few violations
- Small C : Accept more errors for larger margin

The slack variables ξ_i measure how much each point violates the margin constraint. The objective $C \sum_i \xi_i$ penalises these violations, with the parameter C controlling the severity of the penalty.

Geometric picture: Without slack variables, all points must be outside the margin (on the correct side). With slack variables, points can “pay a price” (proportional to ξ_i) to violate this constraint. The parameter C sets the exchange rate between margin size and violations.

Soft-Margin SVM: Dual Formulation

$$\max_{\alpha} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j (x_i^\top x_j)$$

$$\text{subject to } 0 \leq \alpha_i \leq C, \quad \sum_i \alpha_i y_i = 0$$

The only change from hard-margin: upper bound C on the dual variables.

Support vector types:

- $\alpha_i = 0$: Point correctly classified, not on margin
- $0 < \alpha_i < C$: Point exactly on margin ($\xi_i = 0$)
- $\alpha_i = C$: Point inside margin or misclassified ($\xi_i > 0$)

The dual formulation of soft-margin SVM is nearly identical to hard-margin, with only the addition of the upper bound $\alpha_i \leq C$. This elegant result follows from the Lagrangian analysis. The categorisation of support vectors into three types is useful for understanding the solution:

- Points with $\alpha_i = 0$ are “easy”—well inside the correct region
- Points with $0 < \alpha_i < C$ are “on the fence”—exactly on the margin
- Points with $\alpha_i = C$ are “difficult”—inside the margin or misclassified

NB!

Connection to regularisation: The soft-margin SVM objective can be written as:

$$\min_w \frac{1}{n} \sum_{i=1}^n \max(0, 1 - y_i w^\top x_i) + \frac{1}{2C} \|w\|^2$$

This is regularised empirical risk with:

- **Loss:** Hinge loss $\ell(y, f) = \max(0, 1 - yf)$
- **Regularisation:** L2 penalty with $\lambda = 1/C$

Large C (small λ) means less regularisation.

This reformulation reveals that SVMs are simply regularised empirical risk minimisation with a particular loss function (the hinge loss). The hinge loss is zero when the point is correctly classified with margin ≥ 1 , and grows linearly with the margin violation. This contrasts with squared loss (which penalises even correct predictions that are “too confident”) and logistic loss (which always has positive loss).

64.4 Kernel SVMs

Since the SVM dual depends on data only through inner products, we can apply the kernel trick.

Kernel SVM

Replace $x_i^\top x_j$ with $k(x_i, x_j)$:

Dual problem:

$$\max_{\alpha} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j k(x_i, x_j)$$

subject to $0 \leq \alpha_i \leq C, \quad \sum_i \alpha_i y_i = 0$

Prediction:

$$\hat{y}(x) = \text{sign} \left(\sum_{i \in SV} \alpha_i y_i k(x_i, x) + b \right)$$

where $SV = \{i : \alpha_i > 0\}$ is the set of support vectors.

Computing b : Use any support vector with $0 < \alpha_i < C$ (on the margin):

$$b = y_j - \sum_{i \in SV} \alpha_i y_i k(x_i, x_j)$$

The kernel SVM inherits all the benefits of the kernel trick: we can work in infinite-dimensional feature spaces (using e.g. the RBF kernel) while solving a finite-dimensional optimisation problem. Combined with the sparsity of the SVM solution, this makes kernel SVMs practical for many problems.

SVM vs Kernel Ridge Regression

	Kernel SVM	Kernel Ridge
Task	Classification	Regression
Loss	Hinge	Squared
Solution	Sparse (only SVs)	Dense (all points)
Optimisation	Quadratic program	Linear system
Prediction cost	$O(SV)$	$O(n)$

SVMs are preferred when sparsity (fast prediction) matters; kernel ridge is simpler to implement.

The choice between SVM and kernel ridge regression depends on the application:

- **Classification vs regression:** SVMs are designed for classification; kernel ridge regression handles both but is more natural for regression
- **Prediction speed:** SVMs can be much faster when few support vectors exist
- **Implementation complexity:** KRR just requires solving a linear system; SVMs require quadratic programming
- **Probability estimates:** SVMs give hard classifications; KRR gives continuous predictions that can be interpreted as probabilities (with appropriate post-processing)

65 Reproducing Kernel Hilbert Spaces

Section Summary: RKHS Intuition

Every kernel defines a unique function space called an RKHS. Key ideas:

- The kernel $k(x, \cdot)$ is a “feature” that measures similarity to x
- Functions in the RKHS are weighted combinations of these features
- The RKHS norm measures function “complexity” (smoothness)
- Regularisation in RKHS yields the representer theorem

Reproducing Kernel Hilbert Spaces (RKHS) provide the theoretical foundation for kernel methods. While a full treatment requires functional analysis, we can develop useful intuition.

RKHS Definition (Informal)

A **Reproducing Kernel Hilbert Space** (RKHS) \mathcal{H}_k associated with kernel k is a Hilbert space of functions $f : \mathcal{X} \rightarrow \mathbb{R}$ with the **reproducing property**:

$$f(x) = \langle f, k(x, \cdot) \rangle_{\mathcal{H}_k}$$

Interpretation: Evaluating f at x is the same as taking the inner product with the kernel function centred at x .

Key consequences:

- $k(x, x') = \langle k(x, \cdot), k(x', \cdot) \rangle$ — kernel is inner product of “features”
- Functions in \mathcal{H}_k are (infinite) linear combinations: $f = \sum_i \alpha_i k(x_i, \cdot)$

The reproducing property is the defining characteristic of an RKHS. It says that the kernel function $k(x, \cdot)$ (which maps any $x' \mapsto k(x, x')$) acts as a “representer” for point evaluation. To evaluate f at x , we simply take the inner product with this representer.

Intuition: Think of $k(x, \cdot)$ as a “basis function centred at x .” Functions in the RKHS are weighted combinations of these basis functions. The reproducing property says that evaluating f at x extracts the “ x -component” of f .

RKHS Norm and Smoothness

The RKHS norm $\|f\|_{\mathcal{H}_k}$ measures the “complexity” of f .

For $f = \sum_i \alpha_i k(x_i, \cdot)$:

$$\|f\|_{\mathcal{H}_k}^2 = \sum_{i,j} \alpha_i \alpha_j k(x_i, x_j) = \alpha^\top K \alpha$$

Interpretation for RBF kernel:

- Large $\|f\|_{\mathcal{H}_k}$: Function varies rapidly
- Small $\|f\|_{\mathcal{H}_k}$: Function is smooth

Connection to regularisation: Penalising $\|f\|_{\mathcal{H}_k}^2$ encourages smooth functions, explaining why kernel methods generalise well.

The RKHS norm connects directly to the regularisation term in kernel methods. In kernel ridge regression, we minimise:

$$\sum_{i=1}^n (y_i - f(x_i))^2 + \lambda \|f\|_{\mathcal{H}_k}^2$$

The regulariser $\|f\|_{\mathcal{H}_k}^2$ penalises complex functions, and for the RBF kernel, this corresponds to penalising rapid variation. This explains why kernel ridge regression produces smooth predictions—the regulariser explicitly encourages smoothness.

Why RKHS Matters

1. **Representer theorem:** Solutions to regularised problems lie in finite-dimensional subspaces
2. **Kernel choice = prior:** Different kernels encode different notions of smoothness
3. **Infinite dimensions, finite computation:** The kernel trick works because of RKHS structure
4. **Connections:** Links kernel methods to Gaussian processes, Bayesian inference, and functional analysis

The RKHS perspective illuminates several aspects of kernel methods:

- **Why the representer theorem holds:** The regulariser penalises components orthogonal to the span of training kernel functions
- **What the kernel encodes:** Each kernel defines a different RKHS with a different notion of “simple” functions
- **Connection to Gaussian processes:** The RKHS is the “support” of a Gaussian process with covariance function k

66 Related Methods

66.1 K-Nearest Neighbours

While not a kernel method per se, KNN shares the similarity-based philosophy and provides a useful comparison.

KNN Regression

Predict using only the K most similar training points:

$$\hat{y}(\tilde{x}) = \frac{1}{K} \sum_{i \in N_K(\tilde{x})} y_i$$

where $N_K(\tilde{x})$ is the set of K nearest neighbours.

Properties of KNN:

- **Exclusively local:** Only nearby points influence predictions (unlike kernel ridge regression)
- **Simple:** No training phase—just store the data
- **Non-parametric:** Makes no assumptions about functional form
- **Bias-variance tradeoff:**
 - Small K : High variance (sensitive to noise)
 - Large K : High bias (oversmoothing, missing local patterns)

Comparison with kernel methods:

- **KNN:** Hard weighting (equal weight to K neighbours, zero to others)
- **Kernel methods:** Soft weighting (continuous decay with distance)
- **KNN:** No training phase; store data and compute at prediction time
- **Kernel methods:** Training phase (solve linear system or QP); prediction uses learned weights

NB!

KNN uses simple averaging: all K neighbours contribute equally. This ignores the possibility that some neighbours are much closer than others, or that relationships vary across the input space. Kernel methods offer more nuanced weighting.

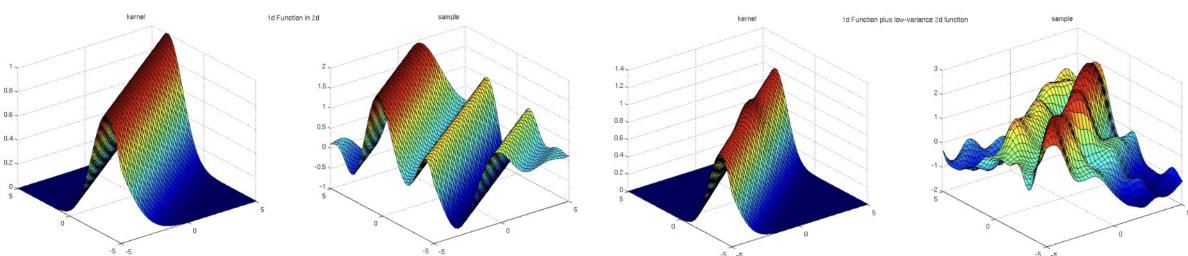


Figure 41: Kernels can capture both global structure and local variation. The kernel induces low-rank global structure (capturing overall trends) while allowing for local adaptation.

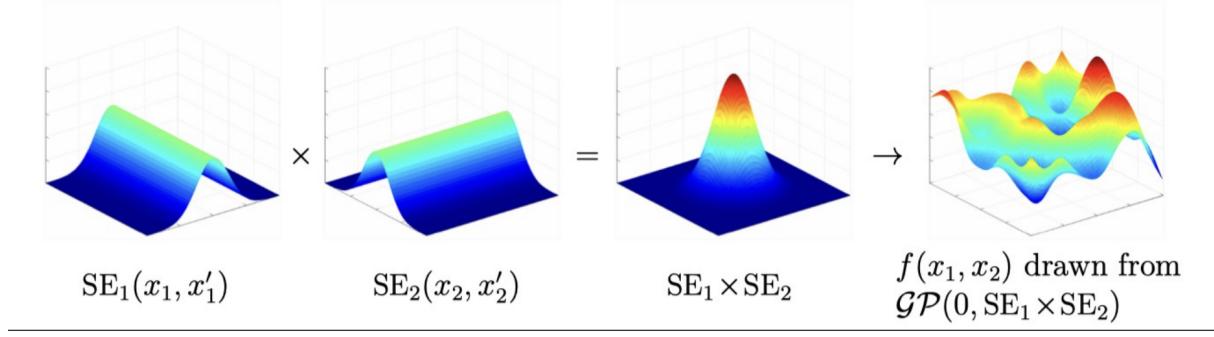


Figure 42: Different features may demand different notions of similarity. By combining or learning kernel parameters, models can discover which features (or combinations) are most indicative of similarity.

Balancing Structure and Flexibility

Effective kernel methods combine:

1. **Low-rank global structure:** Broad trends that apply across the dataset (like PCA)
2. **Local variation:** Fine-grained patterns that differ across regions
3. **Feature-specific similarity:** Different features may contribute differently to similarity

There are many ways to construct custom kernels for your application. More structure (when correct) makes learning easier by reducing the hypothesis space.

67 Practical Considerations

Section Summary: Practical Considerations

- **Kernel selection:** Start with RBF; tune via cross-validation
- **Hyperparameters:** σ (RBF), C (SVM), λ (KRR) — critical for performance
- **Scaling:** $O(n^2)$ storage, $O(n^3)$ training — problematic for large n
- **Approximations:** Random features, Nyström approximation for scalability

67.1 Kernel Selection

Choosing a Kernel

General guidelines:

- **RBF**: Default choice; works well in most situations
- **Linear**: When $d \gg n$ (text, genomics) or data is approximately linear
- **Polynomial**: When interaction effects are known to matter
- **Custom**: When domain knowledge suggests specific similarity structure

Hyperparameter tuning:

- Grid search with cross-validation
- For RBF: $\gamma \in \{2^{-15}, 2^{-13}, \dots, 2^3\}$
- For SVM: $C \in \{2^{-5}, 2^{-3}, \dots, 2^{15}\}$

Data preprocessing:

- **Standardise features**: Critical for RBF (distance-based)
- RBF is not scale-invariant; features on different scales will be weighted differently

The importance of preprocessing cannot be overstated. The RBF kernel computes distances, and if features are on different scales, the kernel will be dominated by the largest-scale feature. Standardising (zero mean, unit variance) puts all features on equal footing.

67.2 Computational Scaling

Computational Complexity

Operation	Time	Space
Compute Gram matrix	$O(n^2d)$	$O(n^2)$
Kernel ridge regression	$O(n^3)$	$O(n^2)$
SVM (typical)	$O(n^2)$ to $O(n^3)$	$O(n^2)$
Prediction (KRR)	$O(n)$ per point	—
Prediction (SVM)	$O(SV)$ per point	—

Bottleneck: For $n = 100,000$, storing K requires 80GB (double precision). Matrix inversion is intractable.

The $O(n^2)$ storage and $O(n^3)$ computation are fundamental limitations of kernel methods. For large datasets, we need approximations.

Scalability Approximations

Random Fourier Features (Rahimi & Recht, 2007):

For shift-invariant kernels (like RBF), approximate:

$$k(x, z) \approx \hat{\phi}(x)^\top \hat{\phi}(z)$$

where $\hat{\phi}(x) \in \mathbb{R}^D$ is a *random* feature map with $D \ll n$.

Procedure:

1. Sample D frequencies ω_j from the kernel's Fourier transform
2. Compute $\hat{\phi}(x) = \sqrt{\frac{2}{D}}[\cos(\omega_1^\top x + b_1), \dots, \cos(\omega_D^\top x + b_D)]$
3. Use standard linear methods on the random features

Complexity: $O(nD)$ instead of $O(n^2)$.

Nyström approximation: Approximate K using a subset of $m \ll n$ landmark points.

Random features are a powerful technique for scaling kernel methods. The key insight is that for shift-invariant kernels, Bochner's theorem guarantees a Fourier representation. By sampling from this representation, we can construct a finite-dimensional approximation to the infinite-dimensional feature space.

NB!

When NOT to use kernel methods:

- **Large n :** When $n > 10,000\text{--}100,000$, consider random features or neural networks
- **Streaming data:** Kernel methods require storing all training data
- **Very high-dimensional features:** Distance becomes meaningless (curse of dimensionality)

68 The Curse of Dimensionality

Kernel methods rely on meaningful notions of distance. In high dimensions, distance becomes problematic.

NB!

In high dimensions, **distance becomes meaningless**: all points become approximately equidistant. This fundamentally limits similarity-based methods.

68.1 Why Distance Fails in High Dimensions

Consider data uniformly distributed in a d -dimensional hypercube:

Volume in High Dimensions

For $X \sim \text{Uniform}(-1, 1)^d$:

- Volume of hypercube: 2^d
- Volume of ϵ -ball: $V_d(\epsilon) = \frac{\pi^{d/2}}{\Gamma(d/2+1)}\epsilon^d$

The fraction of the hypercube within distance ϵ of any point shrinks exponentially with d .

Consequence: To capture a fixed fraction of nearby points, ϵ must grow with d . “Local” methods become global.

This is a fundamental geometric fact. As dimension increases, the volume of a sphere (relative to a cube) shrinks exponentially. Most of the volume of a high-dimensional cube is concentrated near its corners, far from the centre.

What does this mean practically? To capture a fixed fraction of data points as “neighbours,” we need ϵ to grow dramatically with dimension:

d	Volume of domain of X	Volume of sphere	Fraction of data nearby	$\epsilon = \frac{1}{2}$
1	2^1	2ϵ	ϵ	$\frac{1}{2} = 0.5$
2	2^2	$\pi\epsilon^2$	$\frac{\pi}{4}\epsilon^2$	$\frac{\pi}{16} \approx 0.20$
3	2^3	$\frac{4\pi}{3}\epsilon^3$	$\frac{\pi}{6}\epsilon^3$	$\frac{\pi}{48} \approx 0.07$
4	2^4	$\frac{\pi^2}{2}\epsilon^4$	$\frac{\pi^2}{32}\epsilon^4$	$\frac{\pi^2}{512} \approx 0.02$

Figure 43: To capture 10% of data as d increases, ϵ must grow dramatically. In high dimensions, the neighbourhood radius approaches the boundary of the space.

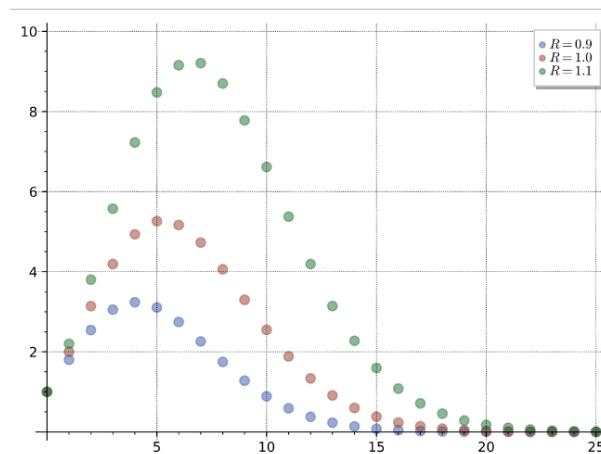


Figure 44: Volume concentration in high dimensions: most volume lies near the boundary, and the concept of “local neighbourhood” becomes meaningless.

Distance Concentration

For points uniformly distributed in high dimensions:

$$\frac{\max_{i,j} \|x_i - x_j\| - \min_{i,j} \|x_i - x_j\|}{\min_{i,j} \|x_i - x_j\|} \rightarrow 0 \quad \text{as } d \rightarrow \infty$$

All pairwise distances become nearly equal. The distinction between “near” and “far” vanishes.

This is the most devastating consequence of high dimensionality for similarity-based methods. When all points are approximately equidistant, there is no meaningful notion of “nearest neighbour” or “most similar.” The kernel matrix approaches a constant matrix, losing all discriminative power.

68.2 Implications for Machine Learning

Implications and Mitigations

- In high dimensions, local methods become global (everything is far)
- Prediction becomes **extrapolation** rather than interpolation
- **Dimensionality reduction** (PCA, autoencoders) can help
- Careful **feature selection** is essential
- For truly high-dimensional data, consider:
 - Linear kernels (scale better)
 - Random features with careful dimension choice
 - Deep learning (learns relevant features)

Remedies:

- **Dimensionality reduction:** PCA, t-SNE, autoencoders can project data to lower dimensions where distance is meaningful
- **Careful feature selection:** Only include features relevant to the prediction task
- **Structured models:** Use domain knowledge to constrain the model (e.g., convolutional structure for images)
- **Deep learning:** Neural networks can learn low-dimensional representations where distance is meaningful

NB!

When NOT to use kernel methods (summary):

- **High-dimensional features:** Distance becomes meaningless; everything is far apart
- **Large n :** The $n \times n$ kernel matrix becomes prohibitively large
- **Streaming/online settings:** Must store all training data

Kernel methods shine with moderate-sized datasets in low-to-moderate dimensions, where domain knowledge can inform kernel choice.

69 Summary

Key Concepts from Kernel Methods

1. **Dual view:** Regression can weight observations by similarity, not just features by coefficients
2. **Kernels:** Functions $k(x, x') = \langle \phi(x), \phi(x') \rangle$ that compute inner products in high-dimensional spaces implicitly
3. **Kernel trick:** Replace $x^\top x'$ with $k(x, x')$ to work in (potentially infinite) feature spaces tractably
4. **Mercer's theorem:** Valid kernels are symmetric and positive semi-definite; they can be combined to build custom similarity measures
5. **Kernel ridge regression:** Solves ridge regression in feature space via an $n \times n$ system; representer theorem guarantees finite representation
6. **Support vector machines:** Maximum-margin classifiers that use only support vectors for prediction; kernel trick enables nonlinear boundaries
7. **RKHS:** Each kernel defines a function space; the RKHS norm measures complexity and connects to regularisation
8. **Common kernels:** Linear (baseline), polynomial (interactions), RBF (infinite-dimensional, local similarity)
9. **Curse of dimensionality:** Distance loses meaning in high dimensions; kernel methods struggle when d is large

Quick Reference: Key Formulas

Kernel definition: $k(x, z) = \langle \phi(x), \phi(z) \rangle$

Common kernels:

$$\text{Linear : } k(x, z) = x^\top z$$

$$\text{Polynomial : } k(x, z) = (x^\top z + c)^M$$

$$\text{RBF : } k(x, z) = \exp\left(-\frac{\|x - z\|^2}{2\sigma^2}\right)$$

Kernel ridge regression: $\hat{y}(\tilde{x}) = k_{\tilde{x}}^\top (K + \lambda I)^{-1} y$

SVM prediction: $\hat{y}(x) = \text{sign}(\sum_{i \in SV} \alpha_i y_i k(x_i, x) + b)$

Representer theorem: $w^* = \sum_{i=1}^n \alpha_i \phi(x_i)$

NB!

When NOT to use kernel methods:

- **Large n :** $O(n^2)$ storage and $O(n^3)$ training become prohibitive
- **High-dimensional features:** Distance becomes meaningless; prefer linear methods or deep learning
- **Streaming/online settings:** Must store all training data

Consider instead: Random features, deep learning, or linear methods with careful feature engineering.



70 Introduction: Why Fairness Matters

Machine learning systems increasingly make or inform consequential decisions about people's lives: who gets a loan, who gets hired, who receives medical treatment, who gets released on bail. When these systems fail, they can cause serious harm to individuals and communities.

Section Summary: The Stakes

- ML systems make high-stakes decisions in hiring, lending, criminal justice, health-care
- Training data encodes historical discrimination—models can learn and amplify bias
- Feedback loops cause predictions to become self-fulfilling prophecies
- Scale and automation mean biased systems affect millions of people

Three features make ML fairness particularly urgent:

1. **Historical biases are encoded in training data.** If past hiring decisions discriminated against women, a model trained on that data will learn to discriminate against women. The model doesn't know the historical context—it simply learns that being female correlates with not being hired.
2. **Automation amplifies bias.** A biased human reviewer might affect hundreds of applications. A biased algorithm can affect millions, consistently applying the same discrimination at scale.
3. **ML systems create feedback loops.** When predictions influence the world, they can generate data that reinforces the original predictions, even if those predictions were initially wrong.

This week introduces the qualitative dimensions of fairness—the conceptual frameworks, types of harm, and sociotechnical considerations that precede any quantitative analysis. Before we can measure fairness (Week 7), we must understand what fairness *means*, what kinds of harm can arise, and why technical solutions alone are insufficient.

71 Machine Learning in Social Context

ML models don't exist in isolation. They operate within institutions, policies, and social structures that shape both their inputs and their impacts. Understanding this broader context is essential for reasoning about fairness.

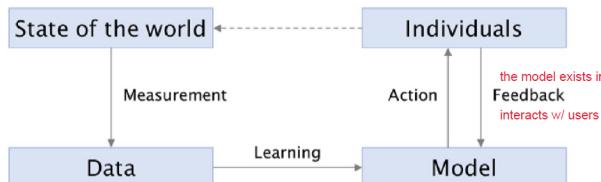


Figure 45: ML systems exist within broader social and institutional contexts.

The figure above illustrates a crucial point: an ML model does not operate in isolation. It sits within a broader context of:

- **Data generation:** Who collected the data? Under what conditions? What was measured and what was omitted?
- **Institutional deployment:** How is the model's output used? Who acts on its predictions?

- **Feedback mechanisms:** How do predictions affect future data collection?
- **Power dynamics:** Who benefits from automation? Who bears the costs of errors?

Sometimes you might not want to include data you consider biased or that is not relevant to the model's intended purpose. But identifying such data requires careful thought about what patterns we want to replicate and which we want to avoid.

71.1 Feedback Loops

NB!

Feedback loops: Predictions can become self-fulfilling prophecies.

Example (Lum & Isaac 2016): Predictive policing systems trained on arrest data send more police to historically over-policed areas, generating more arrests, which reinforces the model's predictions.

The model learns to predict *where police go*, not *where crime occurs*.

Feedback loops are particularly dangerous because they can appear to validate a biased model. If we predict that neighbourhood A is high-crime and send more police there, we will observe more crime in neighbourhood A, making the prediction seem accurate.

The predictive policing example deserves careful unpacking. Consider the chain of reasoning:

1. Police have historically patrolled certain neighbourhoods more intensively (for various historical, political, and resource-allocation reasons)
2. More patrols lead to more observed crime (particularly for offences that require police presence to detect, such as drug possession)
3. Arrest data shows higher crime rates in these neighbourhoods
4. A model trained on arrest data predicts these neighbourhoods are “high crime”
5. Police resources are allocated to these “high crime” areas
6. More patrols lead to more arrests, “confirming” the model’s predictions
7. The cycle continues and intensifies

Other examples of feedback loops:

- **Recommendation systems:** Showing users certain content increases engagement with that content, which trains the system to show more of it
- **Credit scoring:** Denying credit to certain populations means they cannot build credit history, justifying future denials
- **Hiring:** Rejecting candidates from certain backgrounds means they cannot gain experience, reinforcing perceived lack of qualifications

71.2 Language Models and Encoded Bias

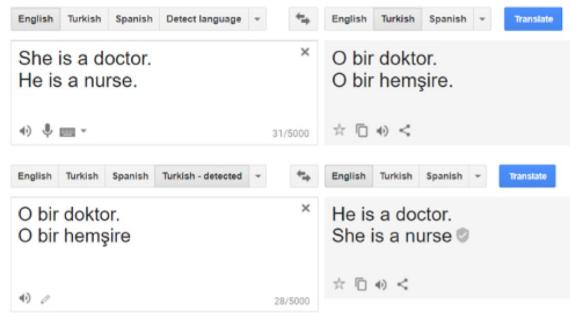


Figure 46: Language models encode societal biases—which patterns should we replicate?

The Turkish language has no gendered pronouns. When translating “O bir doktor. O bir hemşire.” (“They are a doctor. They are a nurse.”), Google Translate produces “He is a doctor. She is a nurse.”—revealing biases learned from training data.

This example is instructive. Turkish uses a gender-neutral pronoun (“o”) for all third-person references. When translating to English, the model must choose a gendered pronoun. It chooses based on statistical patterns in its training data—patterns that reflect historical occupational gender disparities. The model is not “wrong” in a narrow technical sense; it is replicating the patterns it observed.

This raises a fundamental question: *should* ML systems replicate patterns in their training data? The data reflects the world as it is, not necessarily as it should be.

The Reference Distribution Problem

What is the appropriate reference distribution? Should the model:

- **Reflect the world as it currently is** (perpetuating existing disparities)?
- **Reflect the world as it “should” be** (imposing value judgements)?
- **Refuse to make such predictions** (limiting utility)?

There is no technical answer to this question—it requires normative choices about what outcomes we value.

72 Sources of Bias

Bias can enter ML systems at every stage of the pipeline. Understanding where bias originates is essential for addressing it.

Section Summary: Seven Sources of Bias

1. **Historical bias:** Past discrimination encoded in ground truth labels
2. **Representation bias:** Training data doesn't reflect deployment population
3. **Measurement bias:** Features or labels are poor proxies for true quantities
4. **Aggregation bias:** One model for heterogeneous subpopulations
5. **Evaluation bias:** Test data doesn't reflect deployment conditions
6. **Deployment bias:** System used differently than intended
7. **Distribution shift:** Training and deployment populations differ systematically

Comprehensive Bias Taxonomy

1. Historical Bias

Bias arising from the state of the world, even with perfect sampling and measurement.

- Past hiring decisions that discriminated against women
- Criminal justice data reflecting discriminatory policing
- Credit data reflecting historical redlining

This is perhaps the most pernicious form of bias because *no amount of better data collection can eliminate it*. The bias is in the ground truth itself.

What this means: Even if we could perfectly observe and record every historical hiring decision with complete accuracy, the resulting dataset would still be biased because the decisions themselves were discriminatory. The model learns “women don’t get hired” not because the data is flawed, but because that is what actually happened.

2. Representation Bias

Training data fails to represent the population where the model will be deployed.

- Medical trials with predominantly white, male, educated participants
- ImageNet trained mostly on images from Western countries
- Voice recognition trained primarily on American English speakers

Even if the training data is unbiased *for the population it represents*, the model may perform poorly on underrepresented groups.

What this means: A model can be “fair” on its training distribution while being systematically unfair in deployment. If medical AI is trained on data from affluent hospitals, it may fail on patients from under-resourced settings—not because the training data was wrong, but because it was unrepresentative.

3. Measurement Bias

Features or labels don’t accurately capture the underlying construct of interest.

- Using arrest records as a proxy for criminal behaviour (captures policing patterns)
- Using healthcare costs as a proxy for healthcare needs (captures ability to pay)
- Using standardised test scores as a proxy for intelligence (captures test-taking skills and cultural familiarity)

This is a *construct validity* problem: we measure what’s easy to measure, not what we actually care about.

What this means: The gap between what we measure and what we care about is where unfairness enters. From a social science perspective:

$$\text{Arrests} \neq \text{Crimes committed}$$

206

$$\text{Arrests} = f(\text{Crimes committed}, \text{Police presence}, \text{Prosecution decisions}, \dots)$$

The proxy (arrests) conflates the underlying construct (crime) with the measurement

72.1 Feature Omission as a Source of Bias

A distinct but related problem is **feature omission**: failing to measure relevant differences between individuals.

- The model treats different people as identical because distinguishing features weren't collected
- Human decision-makers may capture information not present in the data (context, extenuating circumstances)
- Standardised data collection may systematically miss factors relevant to some groups

Key insight: Human discretion can capture information not in the input form—a human reviewer can have a conversation, observe demeanour, or consider context that was never formalised. Automation loses this ability to incorporate unmeasured information.

NB!

The paradox of protected attributes: Including protected attributes (race, gender) in models is often legally prohibited to prevent discrimination. But *excluding* them can also cause harm:

- Models may use proxy variables that correlate with protected attributes
- Without the attribute, we cannot measure or correct for disparate impact
- Some applications (medical diagnosis) may need protected attributes for accuracy

This creates a genuine dilemma: inclusion risks explicit discrimination, exclusion risks discrimination through proxies while also preventing detection and correction.

73 Types of Harm

ML systems can cause different types of harm, requiring different responses. A foundational taxonomy distinguishes three categories.

Section Summary: Three Types of Harm

- **Allocative harm:** Denies resources or opportunities to individuals
- **Representational harm:** Reinforces stereotypes or subordination
- **Quality-of-service harm:** Worse performance for certain groups

Allocative vs Representational vs Quality-of-Service Harm

Allocative Harm

A system withholds resources, opportunities, or information based on group membership.

Examples:

- Loan denials, hiring decisions, benefit eligibility determinations
- Bail and sentencing recommendations in criminal justice
- Medical resource allocation, insurance pricing

Characteristics:

- Direct, measurable impact on individuals
- Often occurs in high-stakes decisions
- May be legally actionable (disparate impact)

Why it matters: Allocative harm directly affects people's life outcomes—whether they can buy a home, get a job, or receive medical care. These harms are often the focus of anti-discrimination law and are amenable to the quantitative fairness metrics we will develop in Week 7.

Representational Harm

A system reinforces stereotypes, renders groups invisible, or perpetuates subordination.

Examples:

- Image search for “CEO” returning predominantly white male faces
- Language models associating certain professions with genders
- Auto-complete suggesting offensive completions for certain names

Characteristics:

- Shapes perceptions and cultural narratives
- Harder to quantify and measure
- Cumulative effects over time

Why it matters: Representational harm is more subtle but can be equally damaging. It shapes how people think about groups, reinforces social hierarchies, and creates the conditions for allocative harm. A child who sees only white male faces when searching for “scientist” receives a message about who can be a scientist.

Quality-of-Service Harm

A system works less well for certain groups than others.

Examples:

- 208
- Voice recognition with higher error rates for certain accents
 - Facial recognition with lower accuracy for darker skin tones

Cascading Effects Between Harm Types

These three types of harm interact and reinforce each other:

- Stereotypical image search results shape employer perceptions (representational → allocative)
- Biased language models influence hiring algorithms (representational → allocative)
- Underrepresentation in training data leads to worse performance for minority groups (allocative → quality-of-service)
- Quality-of-service disparities create the conditions for allocative harm (lower accuracy → worse decisions)

Representational harm creates the conditions for allocative harm, while allocative harm generates the disparate outcomes that become training data for future representational harm.

74 What is Fairness?

Fairness is not a single concept but a family of related ideas that can conflict with each other. Before measuring fairness quantitatively (Week 7), we must understand its qualitative dimensions.

Three Dimensions of Fairness

1. Legitimacy: Should this system exist at all?

- Precedes discussion of specific harms
- E.g., predicting criminality from faces—is there any legitimate use case?

2. Relative treatment: How does the system allocate resources across groups?

- Can be measured quantitatively (Week 7)
- Demographic parity, equalised odds, calibration

3. Procedural fairness: Is the decision-making process transparent and rational?

- Especially important for complex models
- Explainability, right to reasons

74.1 Legitimacy: The Prior Question

Legitimacy is the foundational question that must be answered before any technical analysis. Some systems should not exist at all, regardless of how carefully they are designed.

NB!**Questions to ask about legitimacy:**

- Is the underlying prediction task scientifically valid? (E.g., predicting “criminality” from faces assumes a relationship that may not exist.)
- Does the system respect human dignity? (E.g., automated emotion detection in job interviews.)
- What are the power dynamics? Who benefits and who is harmed?
- Is automation appropriate for this decision? Some decisions may warrant human judgement regardless of efficiency gains.
- What is the opportunity cost of not building this system? (Sometimes “do nothing” is not a neutral option.)

Legitimacy questions cannot be resolved by technical means. They require ethical reasoning, stakeholder engagement, and democratic deliberation.

74.2 Relative Treatment: Fairness Metrics

Once we have established that a system is legitimate, we can ask how it treats different groups. This is the domain of quantitative fairness (Week 7), where we will see that:

- Multiple reasonable fairness metrics exist
- These metrics generally conflict—you cannot satisfy all of them simultaneously
- Choosing among metrics requires value judgements about what matters

74.3 Procedural Fairness: The Right to Reasons

Even if a decision is “correct” by some outcome measure, it may be unfair if the process is opaque.

Components of Procedural Fairness

- **Transparency:** Can the decision-maker explain how the decision was reached?
- **Consistency:** Are similar cases treated similarly?
- **Contestability:** Can individuals challenge decisions and have them reviewed?
- **Rationality:** Is the decision based on relevant factors?
- **Voice:** Did affected parties have input into the process?

Complex models (especially deep learning) pose challenges for procedural fairness because their reasoning is often opaque even to their designers.

74.4 Individual vs Group Fairness

Two major philosophical approaches to fairness often conflict:

Individual Fairness

Principle: Similar individuals should be treated similarly.

Formalisation: For a metric d measuring similarity between individuals and a metric D measuring similarity between outcomes:

$$d(x_i, x_j) \leq \epsilon \implies D(f(x_i), f(x_j)) \leq \delta$$

Unpacking the notation:

- $d(x_i, x_j)$ measures how “similar” two individuals x_i and x_j are in terms of their features
- ϵ is a threshold defining what counts as “similar”
- $f(x_i)$ is the model’s prediction/decision for individual x_i
- $D(f(x_i), f(x_j))$ measures how different the outcomes are
- δ is a threshold for how different outcomes can be

What this means: If two people are similar (according to d), they should receive similar treatment (according to D). The challenge is defining “similar”—this is where the hard normative work lies.

Advantages:

- Treats people as individuals, not group members
- Aligns with intuitions about personal responsibility
- Avoids arbitrary group boundaries

Challenges:

- Requires defining “similarity”—often contentious
- May perpetuate historical disadvantage if similarity is measured by current status
- Difficult to verify compliance

Group Fairness

Principle: Statistical measures should be equal (or approximately equal) across protected groups.

Common definitions (detailed in Week 7):

- **Demographic parity:** $P(\hat{Y} = 1|A = 0) = P(\hat{Y} = 1|A = 1)$
- **Equalised odds:** Equal true/false positive rates across groups
- **Calibration:** $P(Y = 1|\hat{Y} = p, A = a) = p$ for all groups

Unpacking the notation:

- \hat{Y} is the model's prediction (e.g., "approve loan" = 1, "deny" = 0)
- Y is the true outcome (e.g., "actually repaid" = 1, "defaulted" = 0)
- A is the protected attribute (e.g., race, gender)
- $P(\hat{Y} = 1|A = 0)$ is the probability of a positive prediction given group membership $A = 0$

What demographic parity means: Both groups receive positive predictions at the same rate. If 20% of men are approved for loans, 20% of women should be too.

What equalised odds means: Accuracy is the same across groups—both groups have the same true positive rate (correctly approving good applicants) and false positive rate (incorrectly approving bad applicants).

What calibration means: When you predict "70% chance of repayment," 70% of people with that score should actually repay—and this should hold separately within each group.

Advantages:

- Clear, measurable criteria
- Can detect and correct systematic disparities
- Addresses historical group-level disadvantage

Challenges:

- May conflict with individual fairness
- Requires defining groups—boundaries are often unclear
- Different definitions conflict with each other

NB!

Impossibility Results: Multiple group fairness criteria generally cannot be satisfied simultaneously.

Chouldechova (2017) and Kleinberg et al. (2017) proved that except in degenerate cases, a predictor cannot simultaneously achieve:

- Calibration within groups
- Balance for the positive class (equal true positive rates)
- Balance for the negative class (equal false positive rates)

What this means: These aren't just competing preferences—they are mathematically incompatible. When base rates differ between groups (e.g., if one group has higher default rates), you **cannot** have equal true positive rates, equal false positive rates, and calibration all at once.

This is not a technical limitation to be overcome—it is a fundamental mathematical constraint. **Choosing a fairness criterion is a value judgment, not a technical decision.**

75 Types of Automation

Different types of automation raise different fairness concerns. Understanding *what* is being automated helps identify *which* concerns are most salient.

Three Levels of Automation

Type 1: Automating explicit rules

- **Examples:** Benefits eligibility checking, minimum job requirements, tax calculations
- **What happens:** Rules that already existed (in policy, regulation, or practice) are encoded into software
- **Fairness implication:** Automation makes rules more consistent but loses the flexibility of human discretion
- **Risk:** Edge cases that would have received human consideration are now handled rigidly

Type 2: Automating informal judgements

- **Examples:** Essay grading, medical diagnosis support, credit scoring
- **What happens:** Model learns to mimic expert decisions that were previously made informally
- **Fairness implication:** The model may learn *different* reasoning than the experts intended
- **Risk:** For many decisions, the *process* matters as much as the outcome. A model may achieve similar outcomes through different (potentially objectionable) reasoning

Type 3: Learning rules from data

- **Examples:** Loan approval, predictive policing, hiring recommendation, recidivism prediction
- **What happens:** No pre-existing rules; the model discovers patterns in historical data
- **Fairness implication:** Inherits all biases in the training data; may discover and exploit proxy variables
- **Risk:** Feedback loops; optimising for proxies rather than true objectives; lack of transparency about what is being learned

Automation Types and Fairness Concerns

Type	Primary Fairness Concern
Type 1 (Explicit rules)	Loss of discretion; rigid application to edge cases
Type 2 (Informal judgements)	Process may differ from outcome; learned reasoning may be objectionable
Type 3 (Learning from data)	Bias amplification; feedback loops; proxy exploitation

Type 3 automation is most concerning for fairness because the model may discover patterns that

humans would consider inappropriate or discriminatory. This is the dominant form of modern ML and the most fraught with fairness challenges.

76 Case Studies

Understanding fairness in ML requires examining concrete cases where systems have caused harm. These cases illustrate how abstract bias sources and harm types manifest in practice.

Section Summary: Key Case Studies

- **COMPAS:** Recidivism prediction with racial disparities
- **Amazon hiring:** Gender bias learned from historical data
- **Healthcare algorithms:** Proxy bias underserving Black patients
- **Facial recognition:** Quality-of-service disparities across demographics

76.1 COMPAS: Recidivism Prediction

COMPAS (Correctional Offender Management Profiling for Alternative Sanctions) is a risk assessment tool used in the US criminal justice system to predict likelihood of reoffending.

ProPublica investigation (2016) found:

- Black defendants were nearly twice as likely as white defendants to be labelled high-risk but not reoffend (false positives)
- White defendants were more likely to be labelled low-risk but subsequently reoffend (false negatives)
- Overall accuracy was similar across races, but errors were distributed differently

Northpointe's response: COMPAS is calibrated—among defendants scored as high-risk, similar proportions of Black and white defendants actually reoffended.

The core tension: Both claims are true. This is not a case of one side being wrong; it is a case of different fairness criteria genuinely conflicting.

Unpacking the COMPAS Tension

Let's be precise about what each side is claiming:

ProPublica's claim (equal false positive/negative rates):

- Among people who will *not* reoffend, Black defendants are more likely to be labelled high-risk
- Among people who will reoffend, white defendants are more likely to be labelled low-risk
- This is unfair because innocent Black people pay a higher cost (wrongful detention) than innocent white people

Northpointe's claim (calibration):

- Among people labelled high-risk, similar percentages of Black and white defendants reoffend
- The risk scores mean the same thing regardless of race
- This is fair because a “7” means the same thing whether the defendant is Black or white

Why both can be true: When base rates differ between groups (if one group has higher recidivism rates, which may itself reflect systemic inequality), calibration and equal error rates cannot both hold. This is the impossibility result in action.

The deeper question: Which type of error matters more? Wrongly detaining someone who wouldn't reoffend? Or wrongly releasing someone who would? Different answers lead to different fairness criteria—and there is no purely technical way to resolve this.

NB!

Lesson from COMPAS: “Is this algorithm fair?” is not a well-posed question. We must ask: “Fair according to which definition?” and “Who decides which definition is appropriate?”

76.2 Amazon Hiring Algorithm

Amazon developed an ML system to screen job applicants by learning from historical hiring decisions.

The problem: The system learned to penalise applications containing words like “women’s” (e.g., “women’s chess club captain”) and downgraded graduates of all-women’s colleges.

Why this happened: Historical hiring data reflected existing gender imbalances in tech. The model learned that being male (or proxies for maleness) correlated with being hired.

Attempted fix: Amazon tried removing gender-related features, but the model found other proxies. Eventually, the project was abandoned.

Key insight: Removing protected attributes doesn’t guarantee fairness. Models can learn to discriminate using proxy variables, and in sufficiently rich feature spaces, proxies are nearly impossible to fully eliminate.

Why Removing Protected Attributes Fails

Consider trying to remove gender from a hiring model:

1. Remove the “gender” field—but the model can infer gender from names
2. Remove names—but the model can infer from activities (“women’s rugby team”)
3. Remove activities—but the model can infer from university (all-women’s college)
4. Remove university names—but the model can infer from zip codes
5. And so on...

In high-dimensional feature spaces, there are many ways to reconstruct protected attributes. This is sometimes called the “proxy problem” or “redundant encoding.”

Mathematical intuition: If the protected attribute A is correlated with outcome Y in the training data, and A is correlated with features X_1, X_2, \dots , then the model can learn to predict Y using the features as proxies for A —even if A itself is removed.

76.3 Healthcare Cost Prediction

A widely-used algorithm in US healthcare (studied by Obermeyer et al., 2019) determined which patients qualified for intensive care management programmes.

The problem: The algorithm used predicted healthcare *costs* as a proxy for healthcare *needs*. At the same risk score, Black patients were considerably sicker than white patients.

Why this happened: Black patients historically had less access to healthcare and spent less money on care. Lower spending was interpreted as lower need, when in fact it reflected barriers to access.

Impact: Black patients had to be significantly sicker than white patients to qualify for the same level of care.

NB!**Case Study: Healthcare Risk Prediction**

Context: A widely-used algorithm predicted which patients would benefit from “high-risk care management” programmes.

Design choice: The algorithm used *predicted healthcare costs* as a proxy for *healthcare need*, reasoning that high costs indicate high need.

The problem: Costs depend not only on health status but on access to care:

- Patients who could not afford care had low historical costs
- Patients who faced barriers to access had low historical costs
- These were often the patients with the greatest unmet needs

Result: At a given risk score, Black patients were considerably sicker than white patients with the same score. The algorithm systematically directed resources away from those who needed them most.

Lesson: The proxy (costs) embedded structural inequalities in healthcare access. Optimising for the proxy optimised for something quite different from the intended objective (health needs).

Key insight: This is a *measurement bias* problem. The target variable (costs) was a poor proxy for the construct of interest (needs). The algorithm was working exactly as designed—the design was the problem.

76.4 Facial Recognition Disparities

Multiple studies have found substantial performance disparities in commercial facial recognition systems:

Gender Shades study (Buolamwini & Gebru, 2018):

- Error rates for gender classification were 0.8% for lighter-skinned males vs 34.7% for darker-skinned females
- Commercial systems from IBM, Microsoft, and Face++ all showed similar patterns
- Disparities were not visible in aggregate accuracy metrics

NIST Face Recognition Vendor Test (2019):

- False positive rates varied by factor of 10-100 across demographics
- Asian and African American faces had false positive rates 10-100 times higher than Caucasian faces
- Differences were largest in one-to-many matching scenarios

Why this happens: Predominantly representation bias—training datasets historically over-represented lighter-skinned faces. Also evaluation bias—benchmarks didn’t reveal disparities because they shared the same demographic skew.

Consequences: When deployed in high-stakes contexts (law enforcement, border control), these disparities translate directly into differential rates of false accusations and wrongful detentions.

Why Aggregate Metrics Hide Disparities

Consider a facial recognition system with the following performance:

- 99% accuracy on lighter-skinned faces (80% of test set)
- 70% accuracy on darker-skinned faces (20% of test set)

Aggregate accuracy: $0.8 \times 0.99 + 0.2 \times 0.70 = 0.932$ or 93.2%

The system appears to have excellent performance—93% accuracy!—but this hides a 30% error rate for one group. This is why disaggregated evaluation (breaking down performance by subgroup) is essential.

77 Mitigation Strategies

Addressing fairness in ML requires intervention at multiple stages. This section provides an overview; quantitative methods are covered in Week 7.

Section Summary: Where to Intervene

- **Pre-processing:** Fix the training data
- **In-processing:** Constrain the learning algorithm
- **Post-processing:** Adjust model outputs

Each approach has tradeoffs; no single intervention is universally best.

Mitigation Approaches

Pre-processing: Fix the Data

Modify training data before model training to remove or reduce bias.

Techniques:

- Resampling to balance representation across groups
- Reweighting examples to equalise group importance
- Removing or transforming features that encode protected attributes
- Generating synthetic data to augment underrepresented groups

Advantages: Model-agnostic; addresses root cause *Disadvantages:* May not address all bias sources; can reduce data quality

What this means: If the problem is biased data, fix the data. Oversample underrepresented groups, undersample overrepresented groups, or generate synthetic examples. This works well for representation bias but cannot fix historical bias (which is in the labels themselves).

In-processing: Constrain the Algorithm

Modify the learning algorithm to incorporate fairness constraints.

Techniques:

- Add fairness constraints to the optimisation objective
- Regularisation terms penalising disparity
- Adversarial debiasing (train a model that achieves good accuracy while preventing an adversary from predicting protected attributes)

Advantages: Directly optimises for fairness; can achieve precise guarantees *Disadvantages:* Requires algorithm modification; specific to model type

What this means: Instead of $\min_{\theta} \mathcal{L}(\theta)$ (minimise loss), solve $\min_{\theta} \mathcal{L}(\theta)$ subject to fairness constraints. The model is forced to find solutions that are both accurate and fair (according to the chosen criterion).

Post-processing: Adjust Outputs

Modify model predictions after training to satisfy fairness criteria.

Techniques:

- Threshold adjustment: Use different decision thresholds for different groups
- Calibration: Adjust scores to achieve calibration within groups
- Reject option: Abstain from predictions in uncertain regions

Advantages: Can be applied to any model; doesn't require retraining *Disadvantages:* May not address underlying bias; can feel like a "band-aid"

What this means: Given a trained model, adjust how its outputs are used. If one group has systematically higher scores, lower the²²⁰ threshold for the other group. This achieves demographic parity in outcomes without changing the model itself.

NB!

No free lunch: Achieving fairness typically comes at some cost to overall accuracy. The nature and magnitude of this tradeoff depends on:

- Which fairness criterion is chosen
- How much bias exists in the original data
- The underlying base rates across groups

Stakeholders must decide how to balance these competing objectives.

78 Agency and Recourse

Beyond statistical fairness, individuals affected by ML systems have a stake in understanding and potentially changing decisions about them.

Immutable vs Mutable Characteristics

Models on immutable characteristics (age, race, birthplace):

- Individuals cannot change their fate
- Raises questions of fairness and dignity

Models on mutable characteristics (education, behaviour):

- Obligation to inform individuals how to improve outcomes
- But: Creates opportunity to “game” the system
- Goodhart’s Law: When a measure becomes a target, it ceases to be a good measure

Algorithmic Recourse

Recourse is the ability of an individual to obtain a different outcome by changing their features.

Formal definition: Given individual x with unfavourable outcome $f(x) = 0$, recourse exists if there is an achievable x' such that $f(x') = 1$.

Unpacking the notation:

- x represents all the features describing an individual (income, credit score, employment history, etc.)
- $f(x) = 0$ means the model's decision is negative (loan denied, application rejected)
- x' is a modified version of the individual's features
- "Achievable" means the person can actually make those changes (not "be younger" or "have been born elsewhere")
- $f(x') = 1$ means the model would approve if the features were x'

Requirements for meaningful recourse:

- **Transparency:** Individual knows what factors influenced the decision
- **Actionability:** The factors are things the individual can change
- **Stability:** Changing factors will actually change the outcome
- **Accessibility:** The required changes are feasible for the individual

Example: "Your loan was denied because your credit score is 620. Increase it to 680 and reapply" provides recourse. "Your loan was denied based on 247 factors in a neural network" does not.

NB!

Recourse and model complexity: Black-box models make recourse difficult or impossible. If we cannot explain why a decision was made, we cannot tell individuals how to obtain a different outcome.

This creates tension between:

- Accuracy (complex models often perform better)
- Explainability (simple models are easier to explain)
- Recourse (requires actionable explanations)

Further complications:

- Some features that matter are not actionable (e.g., zip code, age)
- Changing one feature may change model predictions in unexpected ways
- Recourse advice may be technically correct but practically impossible (“increase your income by 50%”)

79 Culpability: Who Is Responsible?

When an automated system makes a harmful decision, who is responsible? This question becomes increasingly important as ML systems mediate more consequential decisions.

The Culpability Question

Multiple parties may bear responsibility for algorithmic harms:

- **Data collectors:** Created or curated the training data
- **Model developers:** Chose the architecture, features, and objective function
- **Deployers:** Decided to use the model for a particular application
- **Operators:** Made individual decisions based on model outputs
- **Regulators:** Failed to establish appropriate oversight

The diffusion problem: When responsibility is distributed across many actors, it becomes difficult to hold anyone accountable. Each party can point to others:

- Data collectors: “We just collected data; we didn’t build the model”
- Developers: “We built a general tool; we didn’t choose how it was used”
- Deployers: “We followed the model’s recommendations; we didn’t design it”
- Operators: “I was just following the system’s output”

The opacity problem: When a model’s reasoning is opaque, it is hard to know whether the harm resulted from:

- Bad data (data collector’s fault)
- Bad model design (developer’s fault)
- Inappropriate deployment (deployer’s fault)
- Misuse of outputs (operator’s fault)

This diffusion and opacity can create “accountability gaps” where harms occur but no one is held responsible.

Implications for Practice

- **Documentation:** Maintain clear records of design decisions and their rationale
- **Audit trails:** Log model inputs, outputs, and human interventions
- **Clear responsibility:** Establish who is accountable for what before deployment
- **Human oversight:** Maintain meaningful human review of consequential decisions
- **Appeal processes:** Provide mechanisms for individuals to challenge decisions

80 Philosophical Considerations

Fairness in ML ultimately rests on philosophical questions that algorithms alone cannot answer.

Section Summary: Deeper Questions

- What does “fair” mean? Different conceptions lead to different criteria
- Who decides? Technical choices encode value judgments
- What are we willing to trade for fairness?

80.1 What is “Fair”?

Different philosophical traditions suggest different fairness criteria:

Libertarian view: Fairness means treating individuals based on their own characteristics and choices, not group membership. Supports individual fairness, calibration.

Implication: A model should predict accurately for each individual based on their features. Group membership should not enter the model. If this leads to disparate outcomes, so be it—the model is treating people as individuals.

Egalitarian view: Fairness means equalising outcomes across groups that have historically been disadvantaged. Supports demographic parity, equalised odds.

Implication: A model should produce similar outcomes across groups. If one group has historically been disadvantaged, the model should not perpetuate that disadvantage. If this requires treating individuals differently based on group membership, that may be justified.

Utilitarian view: Fairness means maximising overall welfare, perhaps with some weight for distribution. May accept some disparities if total outcomes are better.

Implication: A model should maximise some aggregate measure of welfare. If a model that has disparate impact produces better overall outcomes, it may be acceptable—depending on how much weight we give to distribution.

Rawlsian view: Fairness means arranging inequalities to benefit the least advantaged. Suggests prioritising the group with worst outcomes.

Implication: A model should be evaluated by how it affects the worst-off group. Even if a change reduces average accuracy, it may be fair if it improves outcomes for the most disadvantaged. None of these views is obviously correct. The “right” fairness criterion depends on which philosophical framework one adopts.

80.2 Who Decides?

Technical choices in ML systems encode value judgments:

- Which fairness metric to optimise
- How to define protected groups
- How to trade off fairness against accuracy
- What level of disparity is acceptable

These are not purely technical questions. They should involve:

- Domain experts who understand the context
- Affected communities who bear the consequences
- Policymakers who set legal and ethical standards
- Ethicists who can articulate competing values

NB!

The danger of techno-solutionism: Framing fairness as a purely technical problem to be solved by better algorithms obscures the underlying political and ethical questions. “We optimised for equalised odds” is not an answer to “Why are Black defendants given longer sentences?”

80.3 Tradeoffs

Fairness does not exist in isolation. ML practitioners must navigate tradeoffs between:

Fairness vs Accuracy: Achieving fairness often reduces overall predictive accuracy. How much accuracy are we willing to sacrifice?

Different fairness criteria: As impossibility results show, we cannot satisfy all criteria simultaneously. Which violations are more acceptable?

Individual vs group fairness: Treating similar individuals similarly may perpetuate group-level disparities. Equalising group outcomes may treat similar individuals differently.

Short-term vs long-term: Interventions that improve fairness now may have unintended consequences later. Demographic parity in hiring might reduce incentives for groups to acquire qualifications, or might break feedback loops that maintained historical disparities.

Local vs global: A system might be fair within its scope but contribute to broader unfairness. A “fair” loan algorithm operates within a financial system that may be structurally unfair.

81 The Limits of Technical Solutions

A recurring theme in fairness research is that technical solutions alone are insufficient. This is not a counsel of despair but a recognition that fairness is fundamentally a sociotechnical challenge.

Why Technical Fixes Are Insufficient

1. **Fairness is contested:** Reasonable people disagree about what fairness means. No technical definition can resolve normative disagreements.
2. **Context matters:** What counts as “fair” depends on the application, the stakeholders, the history, and the alternatives. There is no universal technical standard.
3. **Metrics conflict:** As we will see in Week 7, reasonable fairness metrics are often mutually incompatible. Choosing among them requires value judgements.
4. **Gaming and adaptation:** People and institutions adapt to algorithmic systems. Technical fixes can be circumvented or may create new problems.
5. **Legitimacy is not technical:** Questions about whether a system should exist, who should control it, and who should benefit from it are political and ethical, not technical.

This does not mean technical analysis is useless. Rather:

- Technical analysis can *reveal* unfairness (auditing, measurement)
- Technical interventions can *mitigate* some types of unfairness
- Technical tools can *support* human decision-making about fairness
- But technical tools cannot *replace* human judgement about values

82 Summary

Key Concepts from Week 6b

1. **Bias amplification:** ML can encode and amplify historical discrimination
2. **Feedback loops:** Predictions can become self-fulfilling prophecies
3. **Construct validity:** Proxies (arrests, costs) conflate what we care about with the measurement process
4. **Seven sources of bias:** Historical, representation, measurement, aggregation, evaluation, deployment, distribution shift
5. **Three types of harm:** Allocative, representational, quality-of-service
6. **Fairness dimensions:** Legitimacy, relative treatment, procedural fairness
7. **Individual vs group fairness:** Similar treatment vs equal statistics—often conflict
8. **Impossibility results:** Multiple fairness criteria cannot be satisfied simultaneously
9. **Mitigation strategies:** Pre-processing, in-processing, post-processing
10. **Recourse:** Individuals should understand decisions and have paths to different outcomes
11. **Culpability:** Responsibility is diffused across data collectors, developers, deployers, operators
12. **Value judgments:** Fairness criteria encode philosophical choices that algorithms cannot resolve

Looking Ahead: Week 7

Week 7 will cover **quantitative fairness**: formal definitions of fairness criteria, mathematical relationships between them, and algorithmic approaches to achieving them. Key topics:

- Formal definitions: demographic parity, equalised odds, calibration
- Three major fairness criteria (independence, separation, sufficiency) and what they mean
- Impossibility results showing these criteria conflict when base rates differ
- How to visualise fairness tradeoffs using ROC curves
- Why “removing” sensitive attributes does not guarantee fairness
- Algorithmic interventions: constrained optimisation, adversarial debiasing
- Measuring and auditing fairness in practice
- The fundamental insight that fairness requires *explicitly encoding values*

The qualitative foundations from this week—understanding types of harm, dimensions of fairness, and the limits of technical solutions—will inform how we interpret and apply those quantitative tools.

83 Introduction: From Qualitative to Quantitative Fairness

Week 6 established that ML systems can cause harm through bias encoded in data and algorithms, identified sources of bias, and distinguished types of harm (allocative, representational, quality-of-service). This week develops the **formal mathematical framework** for measuring and achieving fairness.

The transition from qualitative to quantitative fairness is crucial: while qualitative analysis helps us *identify* potential harms and their sources, quantitative frameworks give us the tools to *measure* fairness, *compare* different approaches, and *enforce* constraints during model training or deployment. However, as we shall see, quantification comes with its own challenges—most notably, the mathematical impossibility of satisfying all reasonable fairness criteria simultaneously.

Chapter Overview

This chapter covers:

- **Classification fundamentals:** Risk scores, thresholds, and evaluation metrics
- **Fairness definitions:** Four main criteria with formal definitions and intuition
- **Impossibility theorems:** Why we cannot have it all
- **ROC analysis:** Fairness as geometric constraints
- **Algorithmic interventions:** Mathematical approaches to debiasing
- **Auditing:** How to measure fairness in practice

Key insight: Fairness criteria are mathematically incompatible. Choosing among them is a *value judgment*, not a technical decision.

84 Classification and Risk Scores

Binary classification learns a function $f : \mathcal{X} \rightarrow \{0,1\}$ that maps feature vectors to class labels. However, most classifiers don't directly output hard decisions—they first produce a **risk score** that estimates the probability of the positive class, which is then thresholded to produce a classification. Understanding this two-stage process (probability estimation followed by thresholding) is essential for understanding fairness, because it reveals where value judgments enter the decision-making process.

Section Summary

- Risk scores estimate $P(Y = 1|X)$ —classification is regression plus thresholding
- Threshold τ converts continuous scores to binary decisions
- Different thresholds yield different tradeoffs between error types
- Cost-sensitive learning makes these tradeoffs explicit

84.1 Risk Scores and Probability Estimation

Risk Scores

A **risk score** $r(x)$ estimates the probability of the positive class:

$$r(x) \approx \mathbb{E}[Y|X = x] = P(Y = 1|X = x)$$

This is regression “hiding” under classification—logistic regression estimates $r(x)$, then we threshold to classify:

$$\hat{y}(x) = \mathbf{1}[r(x) > \tau]$$

where τ is the decision threshold (often 0.5 by default, but this choice is itself consequential) and $\mathbf{1}[\cdot]$ is the indicator function that returns 1 if the condition is true and 0 otherwise.

Unpacking this definition: Let’s break down what each component means:

- **The risk score $r(x)$:** This is the model’s estimate of how likely the positive outcome is for an individual with features x . For example, in a loan default prediction model, $r(x)$ might estimate the probability that an applicant with features x will default on the loan.
- **The conditional expectation $\mathbb{E}[Y|X = x]$:** Since $Y \in \{0, 1\}$, this expectation equals the probability $P(Y = 1|X = x)$. This is because for a binary variable, $\mathbb{E}[Y] = 0 \cdot P(Y = 0) + 1 \cdot P(Y = 1) = P(Y = 1)$.
- **The threshold τ :** This converts a continuous probability into a binary decision. The choice of τ is *not* neutral—it encodes assumptions about the relative importance of different types of errors.
- **The indicator function $\mathbf{1}[\cdot]$:** This mathematical notation simply means “1 if the condition is true, 0 otherwise.” So $\hat{y}(x) = 1$ when $r(x) > \tau$ and $\hat{y}(x) = 0$ otherwise.

84.2 Ideal Model versus Reality

In an **ideal scenario** with perfect knowledge, we could define our classifier’s action as:

$$\hat{y}(x) = \mathbf{1}[\mathbb{E}[Y | X = x] > 0.5] = \begin{cases} 1 & \text{if } \mathbb{E}[Y | X = x] > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

If the expected probability of $Y = 1$ given $X = x$ exceeds 0.5, we predict the positive class; otherwise, we predict 0. This would be the Bayes-optimal classifier under 0-1 loss (when false positives and false negatives are equally costly).

The reality is that we don’t know $\mathbb{E}[Y | X = x]$ *a priori*. We must estimate it from data. This estimation is where regression models are “hiding” under the hood of classification—specifically logistic regression in many binary classification tasks. Logistic regression models the probability that $Y = 1$ as a function of X , providing us with an estimate $\hat{r}(x) \approx \mathbb{E}[Y | X = x]$.

Key Insight: Classification as Thresholded Regression

The risk score is a probability prediction from a regression model. It is the expectation (probability) of the positive class, conditioned on the features. Classification decisions arise from thresholding this continuous probability estimate.

This perspective reveals that binary classification involves *two* separate decisions:

1. **Model choice:** How do we estimate $P(Y = 1|X)$?
2. **Threshold choice:** At what probability do we make a positive prediction?

Both decisions have fairness implications, but the threshold choice is often overlooked.

The threshold τ encodes a **value judgment** about the relative costs of errors. Setting $\tau = 0.5$ implicitly assumes false positives and false negatives are equally costly—rarely true in practice.

Examples of asymmetric costs:

- **Medical screening:** False negatives (missed disease) may be fatal; false positives lead to unnecessary tests. Here we might prefer a low threshold (say $\tau = 0.1$) to catch more cases, accepting more false positives.
- **Spam filtering:** False positives (blocking real email) are more costly than letting spam through. Here we might prefer a high threshold (say $\tau = 0.9$) to avoid blocking legitimate email.
- **Criminal justice:** False positives (wrongful detention) and false negatives (releasing dangerous individuals) have dramatically different costs to different stakeholders. The appropriate threshold depends on whose costs we prioritise.

85 Evaluating Classifiers

Before we can discuss fairness, we need a precise vocabulary for describing how classifiers perform. This section introduces the fundamental metrics that will later be used to define fairness criteria.

Section Summary

- Accuracy treats all errors equally—inappropriate for most real applications
- Cost-sensitive learning assigns explicit costs c_{FP} and c_{FN} to different errors
- ROC curves visualise TPR/FPR tradeoff across all thresholds
- AUC measures threshold-independent discriminative ability

85.1 Accuracy and Its Limitations

Accuracy

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = P(\hat{Y} = Y)$$

where:

- TP = True Positives (correctly predicted positive)
- TN = True Negatives (correctly predicted negative)
- FP = False Positives (incorrectly predicted positive)—Type I errors
- FN = False Negatives (incorrectly predicted negative)—Type II errors

Unpacking accuracy: Accuracy simply counts the proportion of predictions that match the true labels. The formula can be read as: “Of all predictions made, what fraction were correct?” This seems intuitive, but has serious limitations.

NB!

Accuracy has three fundamental problems:

1. **It treats all errors equally.** A false positive and a false negative contribute equally to the accuracy calculation, but their real-world consequences may differ enormously. A cancer screening test that misses a tumour (false negative) has very different consequences than one that triggers an unnecessary biopsy (false positive).
2. **It can be misleading with imbalanced classes.** With 99% negative cases, predicting “negative” always achieves 99% accuracy while being completely useless. This is known as the “accuracy paradox.”
3. **Accuracy for whom?** Different stakeholders bear different costs from different error types. A bank cares about false negatives (defaults); applicants care about false positives (wrongful denials). Accuracy treats these symmetrically, but the people affected do not experience them symmetrically.

85.2 The Confusion Matrix and Error Types

The confusion matrix provides a complete picture of classifier performance by tabulating all four possible outcomes when comparing predictions to truth.

$y = 0$	$y = 1$	
$\hat{y} = 0$	c_{00}	c_{01}
$\hat{y} = 1$	c_{10}	c_{11}

Figure 47: Confusion matrix: the cost matrix c_{ij} weights different outcomes. Rows represent true labels, columns represent predictions. The diagonal contains correct predictions; off-diagonal elements are errors.

Error Rate Definitions

For a classifier with predictions \hat{Y} and true labels Y :

True Positive Rate (TPR) / Sensitivity / Recall:

$$\text{TPR} = P(\hat{Y} = 1|Y = 1) = \frac{TP}{TP + FN}$$

False Positive Rate (FPR) / Fall-out:

$$\text{FPR} = P(\hat{Y} = 1|Y = 0) = \frac{FP}{FP + TN}$$

True Negative Rate (TNR) / Specificity:

$$\text{TNR} = P(\hat{Y} = 0|Y = 0) = 1 - \text{FPR} = \frac{TN}{FP + TN}$$

False Negative Rate (FNR) / Miss rate:

$$\text{FNR} = P(\hat{Y} = 0|Y = 1) = 1 - \text{TPR} = \frac{FN}{TP + FN}$$

Positive Predictive Value (PPV) / Precision:

$$\text{PPV} = P(Y = 1|\hat{Y} = 1) = \frac{TP}{TP + FP}$$

Negative Predictive Value (NPV):

$$\text{NPV} = P(Y = 0|\hat{Y} = 0) = \frac{TN}{TN + FN}$$

Understanding the distinction between rate types: The metrics above fall into two categories based on what we condition on:

- **Conditional on truth** (TPR, FPR, TNR, FNR): These ask “Given we know the true outcome, how likely is a particular prediction?” For example, TPR asks “Among people who actually have the disease, what fraction does the test correctly identify?”

- **Conditional on prediction** (PPV, NPV): These ask “Given a particular prediction, how likely is the true outcome?” For example, PPV asks “Among people who test positive, what fraction actually have the disease?”

This distinction becomes crucial when we discuss different fairness criteria. Some fairness criteria require equal rates *conditional on truth* across groups (separation/equalised odds), while others require equal rates *conditional on prediction* (sufficiency/calibration). These are mathematically different requirements that cannot generally be satisfied simultaneously.

Intuition through medical testing: Consider a medical test:

- **High TPR** (sensitivity): The test catches most cases of disease—few sick people slip through.
- **Low FPR** (high specificity): The test rarely triggers false alarms—few healthy people are incorrectly flagged.
- **High PPV**: When the test is positive, it’s usually right—a positive result is meaningful.

A screening test for a rare disease might have high TPR and low FPR but still have low PPV, because most positive results come from the large pool of healthy people (even a low FPR applied to many people generates many false positives).

85.3 Cost-Sensitive Learning

Rather than treating all errors equally, we can explicitly encode the costs of different outcomes.

Cost-Sensitive Loss

Assign explicit costs to different error types:

$$\mathcal{L}_{\text{cost}} = \frac{1}{n} (FN \times c_{FN} + FP \times c_{FP})$$

This forces you to **explicitly encode values**—what is the relative cost of a false positive versus a false negative?

The optimal threshold depends on the cost ratio. If $c_{FN} = k \cdot c_{FP}$, we should predict positive when:

$$r(x) > \frac{1}{1+k}$$

For example, if false negatives cost twice as much as false positives ($k = 2$), we should use $\tau = 1/3$ rather than $\tau = 0.5$.

Deriving the optimal threshold: The threshold formula $\tau = \frac{1}{1+k}$ arises from decision theory. At the threshold, we should be indifferent between predicting positive and negative. This occurs when:

$$c_{FN} \cdot P(Y = 1|X = x) = c_{FP} \cdot P(Y = 0|X = x)$$

Substituting $P(Y = 0|X = x) = 1 - P(Y = 1|X = x)$ and $c_{FN} = k \cdot c_{FP}$:

$$k \cdot r(x) = 1 - r(x) \implies r(x) = \frac{1}{1+k}$$

So we predict positive when $r(x) > \frac{1}{1+k}$.

Examples:

- $k = 1$ (equal costs): $\tau = 0.5$
- $k = 2$ (FN costs twice FP): $\tau = 1/3 \approx 0.33$
- $k = 9$ (FN costs 9 times FP): $\tau = 0.1$

This approach aims to minimise a weighted sum of errors, allowing for optimisation that reflects actual costs (financial, ethical, health-related) associated with different errors. Libraries like scikit-learn implement this concept through mechanisms like “class weights.”

85.4 ROC Curves

The ROC (Receiver Operating Characteristic) curve is a fundamental tool for visualising classifier performance and understanding fairness constraints.

ROC Curve

The **Receiver Operating Characteristic (ROC)** curve plots TPR against FPR as the threshold τ varies from ∞ to $-\infty$:

- At $\tau = \infty$: Nothing predicted positive $\Rightarrow (FPR, TPR) = (0, 0)$
- At $\tau = -\infty$: Everything predicted positive $\Rightarrow (FPR, TPR) = (1, 1)$
- As τ decreases: Both FPR and TPR increase (generally)

A good classifier has a curve that bows towards the top-left corner (high TPR at low FPR).

Understanding the ROC curve geometrically: The ROC curve traces out the achievable (FPR, TPR) pairs as we vary the threshold. Each point on the curve represents a different operating point—a different tradeoff between catching positives (TPR) and falsely flagging negatives (FPR).

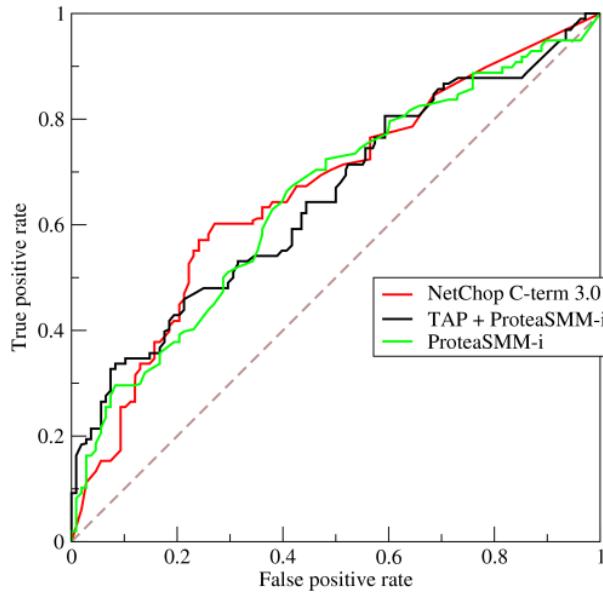


Figure 48: ROC curve. Bottom-left: nothing predicted positive (very high threshold). Top-right: everything predicted positive (very low threshold). Diagonal: random classifier—no better than guessing. The curve bowing toward the top-left indicates discriminative ability. The closer to the top-left corner, the better the classifier.

Key reference points on the ROC curve:

- **(0, 0)**: Predict everything negative—no false positives, but miss all true positives
- **(1, 1)**: Predict everything positive—catch all true positives, but flag all negatives as false positives
- **(0, 1)**: Perfect classifier—catches all true positives with no false positives
- **Diagonal**: Random guessing—TPR = FPR for any threshold

Proper Scoring Rules

A “reasonable” loss function adheres to **Proper Scoring Rules**—criteria ensuring that predicted probabilities accurately reflect true underlying probabilities. Proper Scoring Rules encourage models to estimate true probabilities as accurately as possible (treating the problem like regression), rather than merely optimising for classifications.

Formally, a scoring rule $S(r, y)$ is **proper** if the expected score is maximised when $r(x) = P(Y = 1|X = x)$. Examples include:

- Log loss (cross-entropy): $S(r, y) = y \log r + (1 - y) \log(1 - r)$
- Brier score: $S(r, y) = -(r - y)^2$

If one model’s ROC curve is consistently above another’s across the entire FPR range, it indicates that the former model has a better balance of true positives and false positives for *all* threshold settings. For any proper scoring rule, that model is preferred.

Area Under Curve (AUC)

- AUC = 1: Perfect classifier (exists a threshold achieving TPR=1, FPR=0)
- AUC = 0.5: Random classifier (no better than chance)
- AUC provides threshold-independent evaluation

Probabilistic interpretation: AUC equals the probability that a randomly chosen positive example is scored higher than a randomly chosen negative example. Formally:

$$\text{AUC} = P(r(X^+) > r(X^-))$$

where X^+ is drawn from the positive class and X^- from the negative class.

Model selection strategies:

1. **Threshold-led:** Fix acceptable FPR (e.g., < 20%), choose model with highest TPR in that range
2. **Model-led:** Compare AUC across models, then choose threshold based on costs

Caveat: AUC summarises performance over *all* thresholds, but you typically operate at a single threshold. A model with higher AUC might still perform worse at your specific operating point.

86 Discrimination in Classification

Having established the mechanics of classification, we now turn to how discrimination can arise—and why simply removing protected attributes is insufficient.

Section Summary

- Discrimination can be explicit (using protected attributes) or implicit (using proxies)
- Redundant encoding: combinations of features can reconstruct protected attributes
- Removing protected attributes does *not* guarantee fairness
- This motivates formal fairness criteria that can be measured and enforced

86.1 How Discrimination Arises

A fundamental problem in machine learning fairness is that features X can encode sensitive information about group membership, either directly or indirectly.

Explicit encoding: Features directly encode sensitive attributes (race, gender). When features explicitly encode sensitive information, using these features in a model can lead to discriminatory outcomes. Models may learn to make decisions based on these sensitive attributes, perpetuating or exacerbating existing biases.

Implicit encoding: Features correlate with sensitive attributes—socioeconomic indicators can predict race without using race directly. Models can learn discriminatory patterns through features that are correlated with group membership, even when sensitive attributes are not directly included.

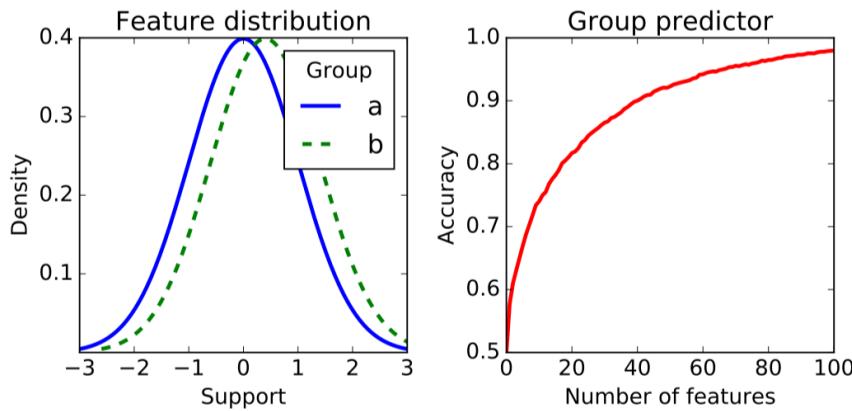


Figure 49: Accumulation of slight predictivity: Groups A and B may be similar on any single feature, making group membership hard to predict from one feature alone. But with many features, each slightly predictive of group membership, we can predict group membership extremely well. The correlation “accumulates” across features.

86.2 Redundant Encodings and the Limits of Attribute Removal

NB!

Redundant encodings: Even if you remove the sensitive attribute, combinations of other features can reconstruct it. Simply dropping race or gender from your model doesn’t guarantee fairness.

In rich feature spaces, proxy variables are nearly impossible to eliminate completely. ZIP code, purchasing patterns, browser choice, first name—each slightly predictive, together highly predictive.

Mathematical intuition: If A is the protected attribute and X_1, \dots, X_p are features, then even if each $\text{Cor}(X_j, A)$ is small, a model trained on all features can achieve high R^2 in predicting A from X . The “curse of dimensionality” works against fairness here.

A set of features, each with only slight predictivity for a sensitive group, can collectively enable a model to classify individuals into groups with high accuracy. This phenomenon means that “fairness through unawareness” (simply not using the protected attribute) is generally insufficient.

86.3 Connection to Week 6: Types of Harm and Metrics

The qualitative harms identified in Week 6 map to different quantitative concerns:

Mapping Harms to Metrics

Allocative harm (denying resources/opportunities):

- Concern: Are positive predictions distributed fairly?
- Relevant metrics: Demographic parity, equalised odds
- Example: Are loans approved at similar rates across groups?

Quality-of-service harm (worse performance for some groups):

- Concern: Are error rates similar across groups?
- Relevant metrics: Equalised odds, equal opportunity
- Example: Does facial recognition have similar accuracy across demographics?

Representational harm (reinforcing stereotypes):

- Harder to quantify with classification metrics
- May require auditing model associations and outputs qualitatively
- Demographic parity can help but doesn't capture all representational concerns

This mapping illustrates that different fairness metrics are appropriate for addressing different types of harm. The choice of metric should be informed by the type of harm we are most concerned about preventing.

87 Quantitative Fairness Criteria

We now formalise the major approaches to quantitative fairness. Each criterion imposes different restrictions on the relationship between the risk score, sensitive attributes, and outcomes.

Section Summary

Four main fairness criteria, each formalising a different intuition:

1. **Demographic parity**: Equal acceptance rates across groups
2. **Equalised odds**: Equal TPR and FPR across groups
3. **Equal opportunity**: Equal TPR across groups (relaxed equalised odds)
4. **Calibration**: Predicted probabilities match actual frequencies within groups

These criteria are *mutually incompatible* when base rates differ between groups.

Notation

Throughout this section:

- $R \in [0, 1]$: Risk score (model output, continuous)
- $\hat{Y} \in \{0, 1\}$: Predicted label (thresholded from R)
- $Y \in \{0, 1\}$: True outcome
- $A \in \{0, 1\}$: Sensitive/protected attribute (e.g., race, gender)

We write $A = 0$ and $A = 1$ for concreteness, but the framework extends to multiple groups. The **base rate** for group a is $P(Y = 1|A = a)$ —the proportion of positive outcomes in that group.

Why base rates matter: Base rates are central to understanding fairness impossibilities. If the base rate differs between groups (e.g., different recidivism rates, different loan default rates), then achieving one fairness criterion necessarily violates others. The base rate difference is often itself a reflection of historical inequities, which raises deep questions about what “fairness” should mean.

87.1 Demographic Parity (Independence)

Demographic Parity / Statistical Parity / Independence

Formal definition:

$$R \perp A \quad \text{or equivalently} \quad \hat{Y} \perp A$$

The notation $R \perp A$ means “ R is independent of A ”—knowing the group membership A tells you nothing about the distribution of the risk score R .

In terms of probabilities:

$$P(\hat{Y} = 1|A = 0) = P(\hat{Y} = 1|A = 1)$$

Relaxed version (bounded disparity):

$$\left| P(\hat{Y} = 1|A = 0) - P(\hat{Y} = 1|A = 1) \right| \leq \epsilon$$

or using the **disparate impact ratio** (80% rule from US employment law):

$$\frac{\min_a P(\hat{Y} = 1|A = a)}{\max_a P(\hat{Y} = 1|A = a)} \geq 0.8$$

Unpacking demographic parity:

- **What it requires:** The probability of receiving a positive prediction must be the same for all groups. If 30% of group A receives loans, then 30% of group B must also receive loans.
- **Independence interpretation:** The risk score distribution $P(R|A = a)$ should be the same for all groups. This is a strong requirement—it says the model should “ignore” group membership entirely in its aggregate behaviour.

- **The 80% rule:** In practice, perfect parity is rarely achieved or required. The disparate impact ratio provides a legal threshold: the acceptance rate for the disadvantaged group should be at least 80% of the rate for the advantaged group.

Intuition: Group membership should not affect the probability of receiving a positive prediction. The classifier should “ignore” the protected attribute in its aggregate behaviour.

When appropriate:

- When we believe base rates *should* be equal (e.g., inherent ability doesn’t differ by group)
- When historical disparities in outcomes reflect discrimination to be corrected
- When the goal is proportional representation (e.g., hiring)

When problematic:

- When base rates genuinely differ and we want predictions to reflect reality
- Can require accepting less qualified individuals from one group over more qualified individuals from another
- Provides no guarantee about error rates or calibration

Achieving Demographic Parity: Orthogonal Projection

To remove the influence of A from predictors:

1. Regress each predictor X_j on A : $X_j = \alpha_j + \beta_j A + \epsilon_j$
2. Use the residuals $\tilde{X}_j = X_j - \hat{\alpha}_j - \hat{\beta}_j A$ as new predictors

The residuals are uncorrelated with A by construction (this follows from the properties of OLS residuals). Training on \tilde{X} removes A ’s linear influence.

Limitations:

- Only removes *linear* correlation; nonlinear proxies may remain
- May substantially reduce predictive power
- Doesn’t guarantee demographic parity if Y itself correlates with A

Mathematical detail: The residual \tilde{X}_j satisfies $\text{Cov}(\tilde{X}_j, A) = 0$ because OLS residuals are orthogonal to the regressors. This is “partialling out” or “orthogonal projection.”

87.2 Equalised Odds (Separation)

Equalised Odds / Separation

Formal definition:

$$R \perp A | Y \quad \text{or equivalently} \quad \hat{Y} \perp A | Y$$

The notation $R \perp A | Y$ means “ R is independent of A *conditional on* Y ”—within each stratum defined by the true outcome, the risk score distribution should not depend on group membership.

The risk score is independent of A **conditional on the true outcome**. This requires:

$$P(\hat{Y} = 1 | A = 0, Y = y) = P(\hat{Y} = 1 | A = 1, Y = y) \quad \text{for } y \in \{0, 1\}$$

Equivalently, **both error rates must be equal across groups**:

$$\text{TPR}_{A=0} = \text{TPR}_{A=1} \quad (\text{equal opportunity for positive class}) \quad (17)$$

$$\text{FPR}_{A=0} = \text{FPR}_{A=1} \quad (\text{equal opportunity for negative class}) \quad (18)$$

Unpacking equalised odds:

- **Conditioning on Y :** We look at people with the same true outcome and ask whether they’re treated similarly regardless of group. Among people who truly defaulted on loans ($Y = 1$), were both groups equally likely to be flagged as high risk? Among people who didn’t default ($Y = 0$), were both groups equally likely to be incorrectly flagged?
- **Two separate requirements:** Equal TPR means equal detection of true positives; equal FPR means equal rates of false alarms. Both must hold.
- **Key difference from demographic parity:** Equalised odds allows different acceptance rates if base rates differ—it only requires equal treatment within each outcome stratum.

Intuition: Conditioning on the true outcome “levels the playing field.” Among people who truly deserve a positive outcome, both groups should have equal chance of receiving one. Among people who truly deserve a negative outcome, both groups should have equal chance of being correctly rejected.

When appropriate:

- When we accept that base rates may differ but want equal treatment conditional on merit
- When both types of errors matter (e.g., criminal justice: wrongful convictions and failures to convict both matter)
- When quality-of-service should be equal across groups

When problematic:

- Requires accurate labels Y —if labels are themselves biased, we optimise for biased accuracy
- May not satisfy demographic parity (different acceptance rates if base rates differ)
- May sacrifice calibration

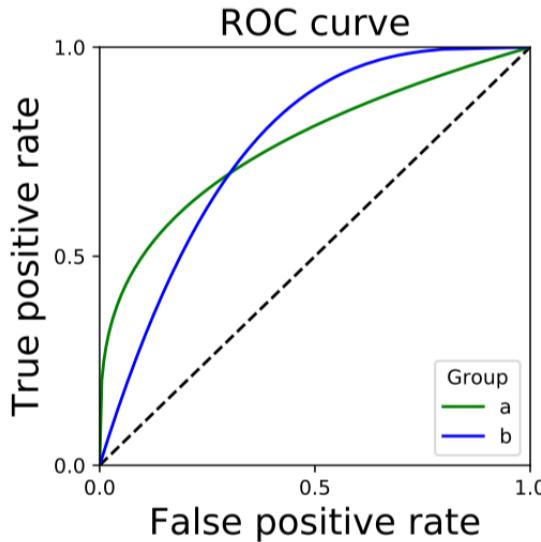


Figure 50: Equalised odds requires operating at the same point on the ROC curve for both groups—same TPR *and* same FPR. If the groups have different ROC curves, this may require using different thresholds for each group.

87.3 Equal Opportunity

Equal Opportunity

A relaxation of equalised odds that requires only equal true positive rates:

$$P(\hat{Y} = 1|Y = 1, A = 0) = P(\hat{Y} = 1|Y = 1, A = 1)$$

Equivalently:

$$\text{TPR}_{A=0} = \text{TPR}_{A=1}$$

This ensures that **qualified individuals have equal opportunity to be selected**, regardless of group membership.

Unpacking equal opportunity:

- **Focus on the positive class:** We only require equal TPR, not equal FPR. This makes sense when the positive outcome is the “desirable” one that we don’t want to withhold unfairly.
- **Weaker than equalised odds:** By not constraining FPR, we have more flexibility in model design. This can lead to better accuracy while maintaining a key fairness property.
- **Asymmetric treatment of errors:** This criterion implicitly values avoiding false negatives more than avoiding false positives (in the fairness sense).

Intuition: Focus on the “positive class” that stands to benefit from positive predictions. Among people who deserve the opportunity (loan, job, parole), group membership shouldn’t affect their chances.

When appropriate:

- When false negatives are the primary fairness concern (missing qualified individuals)

- When the asymmetry of the decision favours focusing on the positive class
- Lending: ensuring creditworthy applicants from all groups have equal access

Limitation: Says nothing about false positives. One group might experience far more false positives (e.g., wrongful accusations), which equal opportunity doesn't address.

87.4 Calibration (Sufficiency)

Calibration / Sufficiency

Formal definition:

$$Y \perp A | R$$

The notation $Y \perp A | R$ means “ Y is independent of A conditional on R ”—once we know the risk score, group membership provides no additional information about the outcome.

Given the same risk score, the probability of a positive outcome is equal across groups:

$$P(Y = 1|R = r, A = 0) = P(Y = 1|R = r, A = 1) = r \quad \text{for all } r$$

A model is **well-calibrated** if predicted probabilities match actual outcome frequencies. **Calibration within groups** requires this to hold separately for each group.

Unpacking calibration:

- **Scores mean the same thing:** If the model says someone has a 70% chance of defaulting, they should actually default 70% of the time—regardless of which group they belong to.
- **Procedural fairness:** Everyone with the same score receives the same treatment. This seems intuitively fair—we're treating “like cases alike.”
- **Note the direction of conditioning:** Calibration conditions on the *prediction*, not the *outcome*. This is the opposite direction from separation/equalised odds.

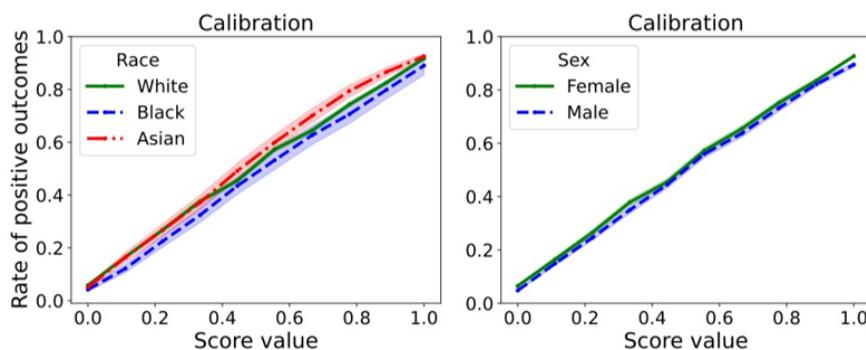


Figure 51: Calibration: a model predicting 70% probability should be correct 70% of the time, for each group separately. A calibration curve plots predicted probabilities against observed frequencies. Perfect calibration is the diagonal line.

Intuition: The risk score means the same thing for everyone. If you score 0.7, you have a 70% chance of the positive outcome, regardless of your group membership. This is a form of **procedural fairness**—everyone with the same score receives the same treatment.

When appropriate:

- When risk scores are used directly (not just for binary decisions)
- When we want scores to have consistent meaning across groups
- When individual accuracy is paramount

When problematic:

- If base rates differ, calibration implies unequal acceptance rates (violates demographic parity)
- May have unequal error rates across groups (violates equalised odds)
- Preserves historical disparities if they are reflected in calibration targets

NB!

Sufficiency (calibration) does not guarantee high individual-level accuracy. A model can be fair in terms of sufficiency—meaning predicted probabilities match observed frequencies for each group—while still making many incorrect individual predictions. Calibration is about aggregate behaviour, not individual precision.

Moreover, calibration can perpetuate discrimination: if historical discrimination causes one group to have worse outcomes (higher default rates due to predatory lending, for example), a calibrated model will reflect these differences, assigning higher risk scores to the disadvantaged group.

The Four Criteria at a Glance

Criterion	Independence	Requires
Demographic Parity	$\hat{Y} \perp A$	Equal acceptance rates
Equalised Odds	$\hat{Y} \perp A Y$	Equal TPR and FPR
Equal Opportunity	$\hat{Y} \perp A Y = 1$	Equal TPR only
Calibration	$Y \perp A R$	Scores mean the same

Each encodes a different intuition about what “fair” means. The choice is fundamentally normative.

Key distinction: Independence and separation condition on different things:

- **Independence:** No conditioning—overall rates should match
- **Separation:** Condition on truth Y —rates within outcome strata should match
- **Sufficiency:** Condition on prediction R —outcomes given the same score should match

87.5 Geometric Interpretation: ROC Space

Fairness criteria can be understood geometrically in ROC space, which provides useful intuition and guides practical implementation.

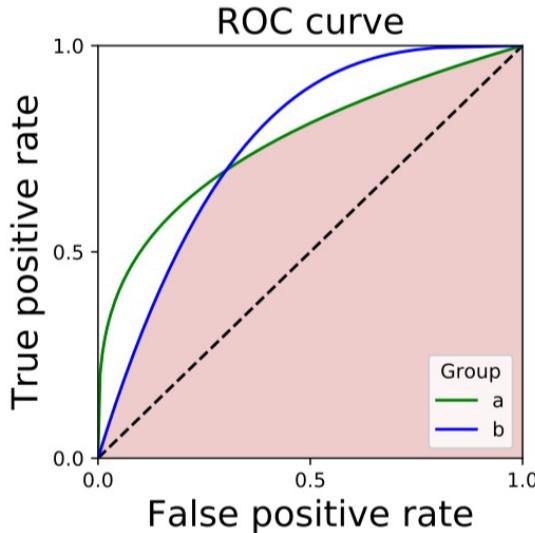


Figure 52: With different base rates, achieving equal error rates requires different thresholds per group. The two groups may have different ROC curves, and satisfying equalised odds requires finding thresholds that map to the same point.

Each group has its own ROC curve (potentially different if the feature-outcome relationship differs between groups). Fairness criteria constrain where we can operate:

Fairness as ROC Constraints

Equalised odds: Both groups must operate at the *same point* (FPR, TPR) in ROC space. If their ROC curves differ, this may require different thresholds per group.

Equal opportunity: Both groups must have the same TPR, but FPR can differ. Graphically: operate at the same height in ROC space.

Demographic parity: More complex—depends on base rates. If base rates equal, same threshold gives demographic parity. If base rates differ, must adjust thresholds to achieve equal acceptance rates.

Calibration: Not directly visualised in ROC space. Requires that the score distributions, when thresholded, yield correct conditional probabilities.

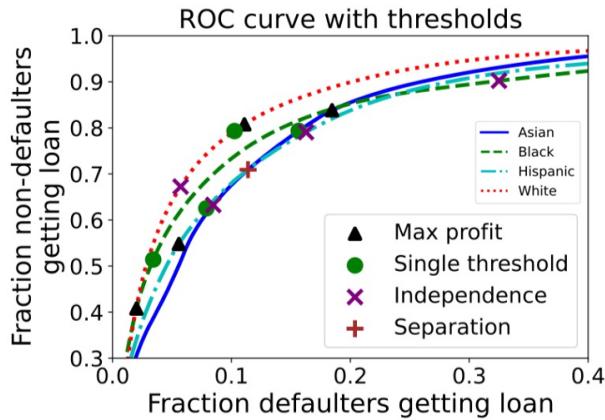


Figure 53: Different fairness criteria lead to different operating points and potentially require different thresholds per group. The “optimal” point depends on which fairness criterion we choose to enforce.

Practical insight: If groups have different ROC curves (which is common when the predictive relationship between features and outcomes differs by group), then:

- A single threshold cannot achieve equalised odds
- Group-specific thresholds are required
- The achievable operating points are constrained by the intersection of what’s achievable for each group

88 Impossibility Theorems

The fairness criteria introduced above represent different, intuitively appealing notions of fairness. A natural question is: can we satisfy multiple criteria simultaneously? The answer, in general, is no—and this is a mathematical fact, not an engineering limitation.

Section Summary

- **Chouldechova (2017):** Calibration + equal FPR + equal FNR impossible unless base rates equal
- **Kleinberg et al. (2016):** Three natural criteria mutually incompatible
 - These are *mathematical facts*, not limitations to be engineered around
 - When base rates differ, we *must* choose which fairness property to sacrifice

88.1 Chouldechova's Impossibility Theorem

Chouldechova's Theorem (2017)

Consider a classifier with:

- Positive predictive value $\text{PPV}_a = P(Y = 1|\hat{Y} = 1, A = a)$
- False positive rate $\text{FPR}_a = P(\hat{Y} = 1|Y = 0, A = a)$
- False negative rate $\text{FNR}_a = P(\hat{Y} = 0|Y = 1, A = a)$

Theorem: If $\text{PPV}_0 = \text{PPV}_1$ (calibration for positive predictions), then:

$$\text{FPR}_0 = \text{FPR}_1 \text{ and } \text{FNR}_0 = \text{FNR}_1 \iff P(Y = 1|A = 0) = P(Y = 1|A = 1)$$

That is, calibration and equal error rates can *only* coexist when the base rates are equal.

What this theorem says in plain language: If your model has the property that a “high risk” prediction means the same thing for both groups (equal PPV = calibration), then you can *only* have equal false positive and false negative rates if the underlying base rates are equal. If one group actually has higher rates of the positive outcome, you’re forced to choose between calibration and equal error rates.

Proof Sketch

Let $p_a = P(Y = 1|A = a)$ be the base rate for group a . By Bayes’ theorem:

$$\text{PPV}_a = \frac{\text{TPR}_a \cdot p_a}{\text{TPR}_a \cdot p_a + \text{FPR}_a \cdot (1 - p_a)}$$

This formula follows from Bayes’ theorem:

$$P(Y = 1|\hat{Y} = 1) = \frac{P(\hat{Y} = 1|Y = 1)P(Y = 1)}{P(\hat{Y} = 1|Y = 1)P(Y = 1) + P(\hat{Y} = 1|Y = 0)P(Y = 0)}$$

Rearranging for FPR:

$$\text{FPR}_a = \frac{\text{TPR}_a \cdot p_a \cdot (1 - \text{PPV}_a)}{\text{PPV}_a \cdot (1 - p_a)}$$

If $\text{PPV}_0 = \text{PPV}_1$ and $\text{TPR}_0 = \text{TPR}_1$ (equal opportunity), then:

$$\text{FPR}_0 = \text{FPR}_1 \iff \frac{p_0}{1 - p_0} = \frac{p_1}{1 - p_1} \iff p_0 = p_1$$

When base rates differ ($p_0 \neq p_1$), the same PPV and TPR imply different FPRs.

Practical implication: In COMPAS (the recidivism prediction tool), if Black and white defendants have different recidivism base rates, we cannot simultaneously achieve:

- Calibration: “High risk” means the same probability of reoffending for both races
- Equal FPR: Equal rates of falsely labelling low-risk defendants as high-risk
- Equal FNR: Equal rates of falsely labelling high-risk defendants as low-risk

This explains the ProPublica/Northpointe dispute: both sides were correct about different fairness criteria. ProPublica found unequal false positive rates; Northpointe showed the tool was calibrated. Both claims were true—and both cannot be “fixed” because they’re mathematically incompatible given unequal base rates.

88.2 Kleinberg, Mullainathan, and Raghavan’s Impossibility Theorem

Kleinberg et al. Impossibility (2016)

Consider three properties for a risk score R :

- 1. **Calibration within groups:** $\mathbb{E}[Y|R = r, A = a] = r$ for all r, a

The expected outcome given a risk score equals that score, for each group separately.

- 2. **Balance for the positive class:** $\mathbb{E}[R|Y = 1, A = 0] = \mathbb{E}[R|Y = 1, A = 1]$

Qualified individuals receive equal average scores across groups.

- 3. **Balance for the negative class:** $\mathbb{E}[R|Y = 0, A = 0] = \mathbb{E}[R|Y = 0, A = 1]$

Unqualified individuals receive equal average scores across groups.

Theorem: Unless $P(Y = 1|A = 0) = P(Y = 1|A = 1)$ (equal base rates), these three properties are mutually incompatible except in degenerate cases (perfect prediction or R constant).

Unpacking this result:

- **Calibration:** Scores should be “honest”—a score of 0.7 means 70% probability.
- **Balance for positives:** Among truly qualified people, both groups should receive similar scores on average. Otherwise, qualified people from one group are systematically underrated.
- **Balance for negatives:** Among truly unqualified people, both groups should receive similar scores on average. Otherwise, unqualified people from one group are systematically overrated.

All three seem reasonable. The theorem says we can’t have all three (unless base rates are equal).

Proof Sketch

Assume calibration holds. Then for group a :

$$\mathbb{E}[R|Y = 1, A = a] = \mathbb{E}[\mathbb{E}[Y|R, A = a]|Y = 1, A = a]$$

By calibration, $\mathbb{E}[Y|R = r, A = a] = r$, so:

$$\mathbb{E}[R|Y = 1, A = a] = \mathbb{E}[R|Y = 1, A = a]$$

Using the law of total expectation and calibration, one can show:

$$\mathbb{E}[R|A = a] = P(Y = 1|A = a)$$

This says: the average risk score for a group equals that group's base rate. **This must hold if the model is calibrated.**

If base rates differ, average scores must differ. But balance for both classes would require average scores conditioned on Y to be equal, which (together with different base rates) implies:

$$\mathbb{E}[R|Y = 1, A = 0] \cdot p_0 + \mathbb{E}[R|Y = 0, A = 0] \cdot (1 - p_0) \neq \mathbb{E}[R|Y = 1, A = 1] \cdot p_1 + \mathbb{E}[R|Y = 0, A = 1] \cdot (1 - p_1)$$

unless the conditional expectations vary to exactly compensate—which they cannot do while maintaining balance for both classes unless $p_0 = p_1$.

88.3 The Independence-Separation-Sufficiency Tradeoffs

The impossibility theorems have important pairwise implications:

The Independence-Sufficiency Tradeoff

Independence ($R \perp A$): Risk score distribution is the same across groups, implying equal selection/acceptance rates.

Sufficiency ($Y \perp A | R$): For any given risk score, outcome probabilities are equal across groups.

When base rates differ:

- **Good calibration \Rightarrow unequal acceptance rates:** If the model accurately reflects differing distributions of risk between groups, acceptance rates will naturally differ
- **Equal acceptance rates \Rightarrow poor calibration:** Forcing equal acceptance rates means the model no longer accurately reflects the true risk distributions

The Independence-Separation Tradeoff

Independence ($R \perp A$): Equal acceptance rates across groups.

Separation ($R \perp A | Y$): Equal error rates (TPR and FPR) across groups.

When base rates differ:

- **Equal acceptance rates \Rightarrow unequal error rates:** The model disregards actual outcome distributions in favour of equalising decision rates, which can amplify disparities
- **Equal error rates \Rightarrow unequal acceptance rates:** The model adjusts predictions to compensate for differences in outcome distributions, leading to different acceptance rates

88.4 Practical Implications of Impossibility

NB!

These impossibility results are **mathematical facts**, not engineering limitations. No algorithm, no matter how sophisticated, can satisfy all fairness criteria when base rates differ.

This means:

- You *must* choose which fairness property to prioritise
- The choice is a **value judgment** requiring normative input
- Technical experts cannot resolve this—it requires ethical and political deliberation
- Different stakeholders may reasonably prefer different criteria

When base rates are equal, all criteria can be satisfied simultaneously. In practice, base rates often differ, forcing difficult choices.

Which Criterion When?

Guidance for choosing among incompatible criteria:

Choose calibration when:

- Risk scores are used directly (not just for binary decisions)
- Individual-level accuracy is paramount
- You believe base rate differences reflect genuine differences in the outcome

Choose equalised odds when:

- Error rates are the primary concern
- Both false positives and false negatives are costly
- Quality-of-service equality is the goal

Choose demographic parity when:

- You believe base rate differences reflect historical discrimination
- Proportional representation is the goal
- You're correcting for measurement bias in labels

Choose equal opportunity when:

- False negatives are the primary fairness concern
- You want to ensure qualified individuals aren't disadvantaged
- False positives are acceptable or less consequential

89 Fairness-Accuracy Tradeoffs

Beyond the impossibility of satisfying multiple fairness criteria simultaneously, there is typically a tradeoff between fairness and predictive accuracy. Understanding this tradeoff is essential for informed decision-making.

Section Summary

- Enforcing fairness typically reduces overall accuracy
- The magnitude of this tradeoff varies—sometimes steep, sometimes mild
- Pareto frontiers visualise the tradeoff: can't improve fairness without sacrificing accuracy (and vice versa)
- The “cost of fairness” depends on how different the groups are

89.1 Quantifying the Tradeoff

When we constrain a model to satisfy a fairness criterion, we typically sacrifice some predictive accuracy. This tradeoff can be visualised as a **Pareto frontier**.

Pareto Frontier for Fairness

Define:

- $\mathcal{A}(f)$: Accuracy (or negative loss) of classifier f
- $\mathcal{F}(f)$: Fairness measure (e.g., $1 - |\text{TPR disparity}|$, so higher is fairer)

The **Pareto frontier** is the set of classifiers where:

- No classifier has both higher accuracy *and* higher fairness
- Improving one metric requires sacrificing the other

Mathematically, f^* is on the Pareto frontier if there exists no f with $\mathcal{A}(f) > \mathcal{A}(f^*)$ and $\mathcal{F}(f) \geq \mathcal{F}(f^*)$ (or vice versa).

Understanding the Pareto frontier:

- **Points on the frontier:** These represent optimal tradeoffs—you cannot improve one objective without worsening the other. Any classifier not on the frontier is “dominated” and should be discarded.
- **The slope of the frontier:** This indicates the “exchange rate”—how much accuracy you sacrifice per unit of fairness gained. Where the frontier is steep, fairness is expensive; where it’s flat, fairness is cheap.
- **Choice along the frontier:** Selecting a point on the frontier is a value judgment. There’s no objectively “best” point—it depends on how much you value fairness relative to accuracy.

Reading a Pareto frontier:

- Points on the frontier represent optimal tradeoffs
- The slope indicates the “exchange rate”—how much accuracy you sacrifice per unit of fairness gained
- Steep regions: fairness is “expensive”
- Flat regions: fairness is “cheap” (can improve fairness with little accuracy loss)

89.2 When is Fairness “Cheap” or “Expensive”?

The cost of fairness depends on several factors:

Determinants of Fairness Cost

Fairness is cheaper when:

- Base rates are similar across groups
- Groups have similar feature distributions
- The protected attribute is weakly predictive of the outcome
- The model has excess capacity (can afford constraints)

Fairness is more expensive when:

- Base rates differ substantially
- Groups have very different feature-outcome relationships
- The protected attribute is highly predictive
- The unconstrained model is already near-optimal

Intuition: If groups are similar, constraining the model to treat them similarly doesn't cost much. If groups are very different (in features, base rates, or the feature-outcome relationship), then forcing equal treatment requires ignoring genuinely predictive information, which costs accuracy.

NB!

A steep fairness-accuracy tradeoff is not necessarily an argument against fairness. It may indicate:

- The “accuracy” being measured includes discriminatory signal
- Historical discrimination created the conditions where A is predictive
- Correcting deep inequities requires real costs

The question is not “Is fairness free?” but “What are we willing to pay for fairness, and who pays?”

Moreover, accuracy itself is not a neutral concept. Whose outcomes count in the accuracy calculation? If the training data reflects historical discrimination, then “accuracy” means “reproducing historical patterns.”

89.3 Multi-Objective Optimisation

In practice, we often care about multiple fairness criteria and accuracy simultaneously. This requires multi-objective optimisation.

Constrained Optimisation Formulation

Fairness as constraint:

$$\max_f \mathcal{A}(f) \quad \text{subject to} \quad \mathcal{F}(f) \geq 1 - \epsilon$$

Find the most accurate classifier among those satisfying the fairness constraint.

Fairness as regularisation:

$$\max_f \mathcal{A}(f) + \lambda \cdot \mathcal{F}(f)$$

Trade off accuracy and fairness via hyperparameter λ . Larger λ means we care more about fairness.

Fairness as Lagrangian:

$$\max_f \min_\mu \mathcal{A}(f) + \mu \cdot (\mathcal{F}(f) - (1 - \epsilon))$$

The dual variable μ represents the “shadow price” of fairness—how much accuracy is sacrificed per unit of fairness gained at the optimum.

Interpreting these formulations:

- **Constraint approach:** You specify a minimum acceptable level of fairness and optimise accuracy subject to that. This is natural when fairness requirements are externally imposed (e.g., legal requirements).
- **Regularisation approach:** You treat fairness as another objective to maximise alongside accuracy. The hyperparameter λ controls the tradeoff and must be chosen based on values.
- **Lagrangian approach:** This is the mathematical dual of the constraint approach. The dual variable μ tells you how “tight” the fairness constraint is—if μ is large, the constraint is binding and relaxing it slightly would yield significant accuracy gains.

90 Algorithmic Interventions

Given the tradeoffs involved, how can we actually build fairer models? Interventions can occur at three stages: before training (pre-processing), during training (in-processing), or after training (post-processing).

Section Summary

Three families of debiasing techniques:

1. **Pre-processing:** Modify training data before model training
2. **In-processing:** Modify the learning algorithm itself
3. **Post-processing:** Modify predictions after training

Each has mathematical formulations and practical tradeoffs.

90.1 Pre-Processing: Modifying the Data

Pre-processing techniques modify the training data to reduce bias before any model is trained. The advantage is that any downstream model inherits the fairness properties; the disadvantage is potential loss of information.

Reweighting (Kamiran & Calders, 2012)

Assign weights to training examples to equalise the weighted label distribution across groups.

For demographic parity, assign weight:

$$w(x, y, a) = \frac{P(Y = y) \cdot P(A = a)}{P(Y = y, A = a)}$$

Unpacking the formula: This weight makes the joint distribution of (Y, A) independent in the weighted data. Examples from underrepresented (group, label) combinations get higher weights; overrepresented combinations get lower weights.

Intuition: Upweight underrepresented (group, label) combinations; downweight overrepresented ones. If group A has fewer positive examples, those positive examples get upweighted.

Effect: A classifier trained on reweighted data will tend toward demographic parity.

Example: If group $A = 1$ has base rate 0.3 and group $A = 0$ has base rate 0.5, positive examples from group 1 will be upweighted and positive examples from group 0 will be downweighted, pushing the model toward equal acceptance rates.

Relabelling / Massaging (Kamiran & Calders, 2009)

Strategically flip labels near the decision boundary:

1. Train a ranker on the original data
2. For the disadvantaged group: flip some negative labels to positive (highest-ranked negatives—those closest to the boundary)
3. For the advantaged group: flip some positive labels to negative (lowest-ranked positives—those closest to the boundary)
4. Balance flips to achieve desired fairness level

Intuition: Correct labels that are most likely to be “wrong” due to historical bias. The examples near the decision boundary are the most ambiguous, so flipping their labels has the least impact on model quality.

Risk: If labels are actually correct, we introduce noise and degrade accuracy. This method assumes that disparities near the boundary reflect bias rather than genuine differences.

Fair Representation Learning (Zemel et al., 2013)

Learn a representation $Z = g(X)$ that:

1. Preserves information useful for prediction: $I(Z; Y)$ high
2. Removes information about the protected attribute: $I(Z; A)$ low

where $I(\cdot; \cdot)$ denotes mutual information.

Optimisation:

$$\min_g \mathcal{L}_{\text{pred}}(g) + \lambda \cdot \mathcal{L}_{\text{fairness}}(g)$$

where $\mathcal{L}_{\text{fairness}}$ penalises Z 's dependence on A .

Advantage: Any downstream model trained on Z inherits the fairness properties. This provides a “fair” feature space that can be used with any classifier.

Challenge: Finding a representation that removes A while preserving Y is difficult when A and Y are correlated. There's an inherent tradeoff.

90.2 In-Processing: Modifying the Algorithm

In-processing techniques modify the learning algorithm to directly optimise for fairness during training. These methods typically provide stronger guarantees but require access to the training process.

Constrained Optimisation

Add fairness constraints to the learning objective:

$$\min_{\theta} \mathcal{L}(\theta) \quad \text{subject to} \quad g_i(\theta) \leq 0 \quad \text{for fairness constraints } i$$

Example (equalised odds constraint):

$$g_1(\theta) = |\text{TPR}_0(\theta) - \text{TPR}_1(\theta)| - \epsilon \tag{19}$$

$$g_2(\theta) = |\text{FPR}_0(\theta) - \text{FPR}_1(\theta)| - \epsilon \tag{20}$$

Challenge: Fairness constraints often involve indicator functions (non-differentiable). The TPR and FPR depend on discrete predictions, making gradient-based optimisation difficult.

Common solutions:

- Use surrogate losses (e.g., logistic loss approximates 0-1 loss)
- Use constrained optimisation solvers (e.g., CVXPY for convex problems)
- Use Lagrangian relaxation with iterative updates

Adversarial Debiasing (Zhang et al., 2018)

Train two networks jointly:

1. **Predictor** f : Maps features X to predictions \hat{Y}
2. **Adversary** g : Maps predictions \hat{Y} (and possibly Y) to protected attribute A

Optimisation (minimax game):

$$\min_f \max_g \mathcal{L}_{\text{pred}}(f) - \lambda \cdot \mathcal{L}_{\text{adv}}(g, f)$$

The predictor tries to:

- Minimise prediction loss (be accurate)
- Maximise adversary's loss (make A unpredictable from \hat{Y})

For demographic parity: The adversary tries to predict A from \hat{Y} . If it cannot, then $\hat{Y} \perp A$.

For equalised odds: Give adversary access to both \hat{Y} and Y . The adversary should be unable to predict A from (\hat{Y}, Y) , which means $\hat{Y} \perp A | Y$.

Intuition: If the adversary cannot recover A from predictions (possibly conditioning on Y), the predictions satisfy the desired independence. The predictor learns to “hide” group membership from its predictions.

Fairness-Aware Regularisation

Add a regularisation term penalising unfairness:

$$\min_{\theta} \mathcal{L}(\theta) + \lambda \cdot \mathcal{R}_{\text{fair}}(\theta)$$

Example regularisers:

- Covariance penalty: $\mathcal{R} = |\text{Cov}(\hat{Y}, A)|$ —penalises correlation between predictions and group
- Mutual information: $\mathcal{R} = I(\hat{Y}; A)$ (or conditional versions)—penalises information shared between predictions and group
- Disparity penalty: $\mathcal{R} = (\text{TPR}_0 - \text{TPR}_1)^2 + (\text{FPR}_0 - \text{FPR}_1)^2$ —directly penalises error rate differences

Trade-off: λ controls the fairness-accuracy tradeoff. Higher λ gives more weight to fairness at the cost of accuracy.

90.3 Post-Processing: Modifying Predictions

Post-processing techniques adjust a trained model’s outputs to achieve fairness, without retraining. These are useful when you cannot retrain the model (e.g., using a third-party API) or want

to quickly adjust an existing deployment.

Threshold Adjustment (Hardt et al., 2016)

Use **group-specific thresholds** to achieve equalised odds:

$$\hat{Y}_a = \mathbf{1}[R > \tau_a]$$

Algorithm:

1. Plot ROC curves for each group separately
2. Find thresholds (τ_0, τ_1) that achieve the same (FPR, TPR) point
3. If no exact match exists, use randomisation: with probability p , use one threshold; otherwise use another

Optimisation (for equalised odds):

$$\max_{\tau_0, \tau_1} \text{Accuracy} \quad \text{s.t.} \quad \text{TPR}_0(\tau_0) = \text{TPR}_1(\tau_1), \quad \text{FPR}_0(\tau_0) = \text{FPR}_1(\tau_1)$$

Advantage: Works with any pre-trained model (black-box). No access to training needed.

Disadvantage: May require very different thresholds, reducing overall accuracy substantially. Also raises questions about treating individuals differently based on group membership.

Calibrated Equalised Odds (Pleiss et al., 2017)

Achieve equalised odds while maintaining calibration (as much as possible) by finding the optimal randomised threshold policy.

Insight: Post-processing can achieve any point in the convex hull of achievable (FPR, TPR) pairs via randomisation. If we can't hit exactly (FPR^*, TPR^*) with a deterministic threshold, we can mix between two thresholds probabilistically to achieve it in expectation.

Constraint: Calibration is partially preserved if we only randomise between adjacent thresholds (thresholds that correspond to adjacent points on the ROC curve).

Reject Option Classification

Abstain from predictions in uncertain regions, especially where fairness violations are likely:

$$\hat{Y}(x) = \begin{cases} 1 & \text{if } R(x) > \tau_+ \\ 0 & \text{if } R(x) < \tau_- \\ \text{defer to human} & \text{otherwise} \end{cases}$$

Set τ_+ and τ_- per group to balance error rates.

Advantage: Avoids harmful predictions in uncertain cases. Recognises that some decisions shouldn't be automated.

Disadvantage: Requires human reviewers for deferred cases; may defer disproportionately for some groups (which itself could be a fairness concern).

Comparison of Intervention Approaches

Stage	Advantages	Disadvantages
Pre-processing	Model-agnostic Addresses root cause	May lose information Requires access to data
In-processing	Direct optimisation Precise guarantees	Model-specific Requires retraining
Post-processing	Black-box compatible No retraining needed	Band-aid solution May degrade accuracy

Rule of thumb: Pre-processing is appropriate when you control the data pipeline; in-processing when you control the training process and want strong guarantees; post-processing when you're working with a fixed model or need quick adjustments.

91 Evaluation and Auditing

Building fair models is only part of the challenge. We also need methods to evaluate and audit models for fairness, both before deployment and during operation.

Section Summary

- Fairness auditing requires disaggregated evaluation by group
- Multiple metrics should be computed—no single metric suffices
- Statistical uncertainty: fairness metrics have confidence intervals
- Intersectionality: consider subgroups defined by multiple attributes

91.1 Auditing Procedure

Fairness Audit Checklist	
1. Define protected groups	<ul style="list-style-type: none">• Which attributes are protected in this context?• How are groups defined? (Self-reported? Inferred? Administrative?)• Consider intersectionality (e.g., Black women vs white women vs Black men)
2. Choose appropriate metrics	<ul style="list-style-type: none">• What type of harm are we most concerned about?• Which fairness criteria align with stakeholder values?• Compute multiple metrics—they may tell different stories
3. Compute metrics with uncertainty	<ul style="list-style-type: none">• Point estimates are not enough—report confidence intervals• Small subgroups have high variance• Consider bootstrap or Bayesian approaches
4. Compare to baselines	<ul style="list-style-type: none">• How does this model compare to alternatives?• What would a “fair” model look like?• What is the status quo (human decision-makers)?
5. Document and communicate	<ul style="list-style-type: none">• Which criteria were evaluated?• Which were satisfied/violated?• What tradeoffs were made and why?

91.2 Statistical Considerations

Fairness metrics are *estimates* computed from finite samples. They have uncertainty, and this uncertainty can be substantial for small subgroups.

Confidence Intervals for Fairness Metrics

For rate-based metrics (TPR, FPR, acceptance rate), use standard binomial confidence intervals.

Example: Estimating TPR disparity $\Delta = \text{TPR}_0 - \text{TPR}_1$

If $\hat{\text{TPR}}_a = \frac{TP_a}{TP_a + FN_a}$ with $n_a = TP_a + FN_a$ positive examples in group a :

$$\text{SE}(\hat{\text{TPR}}_a) \approx \sqrt{\frac{\hat{\text{TPR}}_a(1 - \hat{\text{TPR}}_a)}{n_a}}$$

This is the standard error for a proportion, derived from the binomial distribution.

For the difference (assuming independence between groups):

$$\text{SE}(\hat{\Delta}) = \sqrt{\text{SE}(\hat{\text{TPR}}_0)^2 + \text{SE}(\hat{\text{TPR}}_1)^2}$$

A 95% confidence interval: $\hat{\Delta} \pm 1.96 \cdot \text{SE}(\hat{\Delta})$

Interpretation: If this interval contains 0, we cannot conclude the TPRs are different at the 5% significance level.

NB!

Small subgroups are problematic. If a protected group has few examples:

- Confidence intervals will be wide
- Point estimates may be unreliable
- Cannot detect fairness violations with statistical significance

Intersectional subgroups (e.g., elderly Black women) may be very small even in large datasets. This is a fundamental challenge—the groups most at risk of discrimination may be the hardest to audit.

Practical implication: A model might appear fair simply because we lack the statistical power to detect unfairness in small subgroups. Absence of evidence is not evidence of absence.

91.3 Multiple Metrics and Their Relationships

Metric Relationships

Given the impossibility results, a model cannot satisfy all metrics simultaneously (unless base rates are equal). Observing:

Calibration satisfied, equalised odds violated: The model is accurate but has disparate error rates. Common when base rates differ. The model treats the groups “fairly” in terms of score meaning but unfairly in terms of error distribution.

Demographic parity satisfied, calibration violated: The model achieves equal acceptance rates but scores mean different things for different groups. May be appropriate if base rate differences reflect historical discrimination.

All metrics approximately satisfied: Either base rates are similar, or the model is close to random (not useful).

Use multiple metrics to understand the full picture:

- Demographic parity / disparate impact ratio
- TPR and FPR by group (for equalised odds)
- Calibration curves by group
- PPV and NPV by group

91.4 Intersectionality

Intersectional Fairness

Standard fairness analysis considers protected attributes separately. But individuals belong to multiple groups, and harms may concentrate at intersections.

Example (Gender Shades study, Buolamwini & Gebru 2018): Facial recognition error rates:

	Male	Female
Lighter skin	0.8%	7.0%
Darker skin	12.0%	34.7%

Aggregate statistics by gender or by skin tone would miss that *darker-skinned women* experience dramatically worse performance.

The problem: Looking at gender alone, you might see Female error rate $\approx 21\%$ and Male error rate $\approx 6\%$ —concerning but not catastrophic. Looking at skin tone alone, you might see Darker $\approx 23\%$ and Lighter $\approx 4\%$. But the intersection (darker-skinned women at 35%) is much worse than either marginal statistic suggests.

Recommendation: Compute fairness metrics for intersectional subgroups, not just marginal groups. But beware: smaller subgroups mean higher variance.

92 Fairness is Not a Technical Problem

We conclude by emphasising the fundamental point: while this chapter has provided technical tools for measuring and enforcing fairness, the choice of which fairness criterion to use is not a technical decision.

Section Summary

- Quantitative fairness criteria conflict—you must choose
- The choice encodes values that algorithms cannot determine
- Different criteria favour different stakeholders
- “Fair ML” is not a product but a sociotechnical process

Different Criteria, Different Models

The same data, with different fairness criteria, yields different models:

Max profit / unconstrained: No fairness constraints; potentially wildly different rates by group. Optimises pure accuracy (or profit).

Single threshold: One threshold for all groups; achieves calibration but may violate equalised odds if groups have different ROC curves.

Demographic parity: Equal acceptance rates; may sacrifice calibration and equalised odds. Treats groups equally in terms of outcomes.

Equalised odds: Equal error rates; may sacrifice calibration and demographic parity. Treats groups equally in terms of accuracy.

Each choice advantages some stakeholders and disadvantages others. This is an irreducibly political decision.

92.1 POSIWID: The Purpose of a System is What it Does

“The Purpose of a System is What it Does” — Stafford Beer (management consultant and cybernetician)

When dealing with complex systems, focus on the results they generate, not their stated intentions.

Don’t focus narrowly on “the algorithm.” Evaluate the larger system:

- What outcomes does it produce in practice?
- Who benefits and who is harmed?
- What feedback loops does it create?
- How does it interact with human decision-making?
- What institutional incentives shape its use?

A technically “fair” algorithm can still produce unfair outcomes if embedded in an unfair system. Conversely, addressing algorithmic bias without addressing systemic issues may have limited impact.

NB!

Technical fairness metrics, while useful, cannot capture the full complexity of fairness in social contexts. A model that satisfies a formal fairness criterion may still cause harm when deployed in practice. Context matters: the same model might be appropriate in one setting and harmful in another.

Key questions to ask:

- Is the outcome being predicted the right outcome to predict?
- Are the labels we're training on themselves biased?
- Who has power over the system's design and deployment?
- Are affected communities involved in decisions about fairness criteria?

9.2.2 Returning to Week 6: The Full Picture

The quantitative tools in this chapter operationalise the qualitative concerns from Week 6:

Connecting Qualitative and Quantitative

Historical bias (Week 6) → Calibration preserves it; demographic parity may correct it

Measurement bias (Week 6) → All metrics can be misleading if Y is a poor proxy for what we actually care about

Allocative harm (Week 6) → Demographic parity addresses disparate allocation

Quality-of-service harm (Week 6) → Equalised odds addresses disparate error rates

Feedback loops (Week 6) → Static fairness metrics don't capture dynamic effects; a "fair" model at deployment may become unfair as it influences the data it's trained on

Impossibility results (Week 6 mention) → Now proven formally (Chouldechova, Kleinberg et al.)

The qualitative analysis helps you decide *which* quantitative criterion is appropriate. The quantitative analysis helps you *measure and enforce* that criterion.

93 Summary

Key Concepts from Week 7

1. **Risk scores:** Classification uses regression to estimate $P(Y = 1|X)$, then thresholds. The threshold encodes value judgments about error costs.
2. **ROC curves:** Visualise TPR/FPR tradeoff across thresholds; fairness constrains operating points. AUC provides threshold-independent evaluation.
3. **Four fairness criteria:**
 - Demographic parity: $P(\hat{Y} = 1|A = 0) = P(\hat{Y} = 1|A = 1)$ —equal acceptance rates
 - Equalised odds: Equal TPR and FPR across groups—equal error rates
 - Equal opportunity: Equal TPR across groups—qualified individuals treated equally
 - Calibration: $P(Y = 1|R = r, A = a) = r$ for all groups—scores mean the same thing
4. **Impossibility theorems:** Calibration + equal error rates impossible unless base rates equal. This is mathematics, not engineering.
5. **Fairness-accuracy tradeoffs:** Pareto frontiers quantify the cost of fairness. The cost depends on how different the groups are.
6. **Interventions:** Pre-processing (reweighting, relabelling, fair representations), in-processing (constrained optimisation, adversarial training, regularisation), post-processing (threshold adjustment, reject option)
7. **Auditing:** Compute multiple metrics with confidence intervals; consider intersectionality. Small subgroups pose statistical challenges.
8. **Values matter:** Choosing a fairness criterion is a normative decision, not a technical one. The impossibility results force us to make value-laden choices.

Practical Guidance

When building a model:

1. Identify protected groups and potential harms (Week 6 framework)
2. Choose fairness criterion(s) appropriate to the context and stakeholder values
3. Train with appropriate intervention (or use post-processing)
4. Audit with multiple metrics, including confidence intervals
5. Document tradeoffs and justify choices

When auditing a model:

1. Compute disaggregated metrics by group
2. Check multiple fairness criteria
3. Report confidence intervals, especially for small subgroups
4. Consider intersectional subgroups
5. Compare to baselines (other models, human decision-makers)

NB!

The impossibility results are not a counsel of despair. They clarify the choices we face:

- We cannot satisfy everyone simultaneously
- We must be explicit about whose interests we prioritise
- Technical tools can enforce our choices, but cannot make them for us

Fairness in ML is ultimately about **power**: who decides what “fair” means, and who bears the costs of our choices?

The mathematical framework in this chapter is necessary but not sufficient for building fair systems. It must be combined with stakeholder engagement, ongoing monitoring, and willingness to revise decisions as we learn more about a system’s real-world impacts.

94 Decision Trees

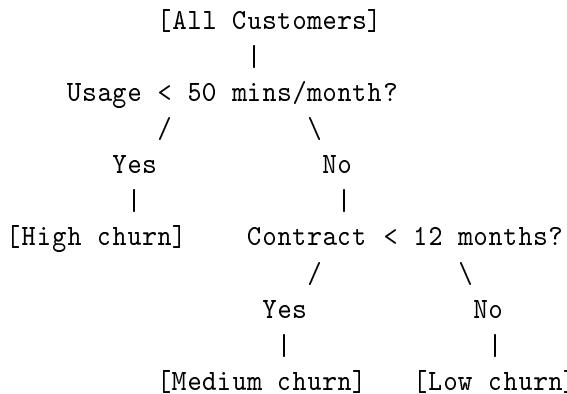
Chapter Summary

Decision trees recursively partition the feature space using axis-aligned splits, predicting a constant value within each region. They offer **interpretability** (rules as flowcharts), **flexibility** (handle mixed feature types, nonlinear patterns), and **automatic feature selection**—but suffer from **high variance** and **suboptimal accuracy**. Key concepts: greedy recursive partitioning via **Gini impurity** or **entropy** (classification) and **MSE reduction** (regression); **cost-complexity pruning** to prevent overfitting; and **ensemble methods** (bagging, random forests) that combine many trees to reduce variance while preserving flexibility. Trees are piecewise constant approximations—simple individually but powerful when combined.

Decision trees are fundamental building blocks for constructing prediction functions in both classification and regression settings. Unlike methods that assume a particular functional form (linear regression assumes linearity, logistic regression assumes log-odds linearity), decision trees make *no assumptions* about the relationship between features and the response. Instead, they learn this relationship directly from the data by partitioning the feature space into regions and making simple predictions within each region.

94.1 Intuition: Recursive Binary Partitioning

A decision tree learns by repeatedly asking “yes/no” questions about features, splitting the data into progressively smaller regions until each region is sufficiently homogeneous. Consider predicting whether a customer will churn:



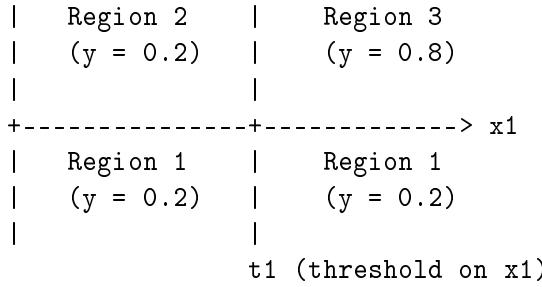
Each internal node represents a *split*—a test on a single feature. Each leaf represents a *region*—a subset of feature space where we make a constant prediction. The path from root to leaf defines a conjunction of conditions: customers with usage ≥ 50 mins AND contract ≥ 12 months fall into the “low churn” region.

What this structure captures: The tree naturally encodes *interaction effects*. The effect of contract length on churn depends on usage level—we only ask about contract length for customers with usage ≥ 50 mins. This conditional structure is precisely what makes trees powerful for capturing complex, non-additive relationships that linear models would miss without explicit feature engineering.

This recursive partitioning produces axis-aligned rectangular regions in feature space. For two features (x_1, x_2) , a tree with two splits might produce:

x2

~



Key insight: Trees are fundamentally *piecewise constant* models. Within each region, the prediction is a single value—the mean (regression) or mode (classification) of training points in that region. Flexibility comes not from the complexity of the prediction function within regions, but from the number and placement of region boundaries. This is analogous to approximating a curve with a histogram: simple flat segments that collectively capture complex shapes.

94.2 Tree Structure and Terminology

Before proceeding to the mathematics, let us establish precise terminology. A decision tree is a directed acyclic graph with a specific hierarchical structure.

Tree Components

A decision tree T consists of:

Nodes:

- **Root node:** Contains all training data; top of the tree
- **Internal nodes:** Define splits; have exactly one parent and two children (for binary trees)
- **Leaf nodes** (terminal nodes): Define regions; contain predictions

Edges: Connect parent to children, labelled by split outcome (e.g., $x_j \leq t$ vs $x_j > t$)

Depth: Length of the longest path from root to any leaf

Split: At internal node with dataset \mathcal{D} , a split (j, t) partitions data into:

$$\mathcal{D}_L = \{(x_i, y_i) \in \mathcal{D} : x_{ij} \leq t\}, \quad \mathcal{D}_R = \{(x_i, y_i) \in \mathcal{D} : x_{ij} > t\}$$

where j is the feature index and t is the threshold.

Unpacking the split notation: The expression $\mathcal{D}_L = \{(x_i, y_i) \in \mathcal{D} : x_{ij} \leq t\}$ reads as “the left child dataset \mathcal{D}_L consists of all feature-response pairs (x_i, y_i) from the current dataset \mathcal{D} such that the j -th feature of x_i (written x_{ij}) is less than or equal to threshold t .” The subscript ij means “observation i , feature j .” This notation becomes natural once you think of the data as a matrix with observations as rows and features as columns.

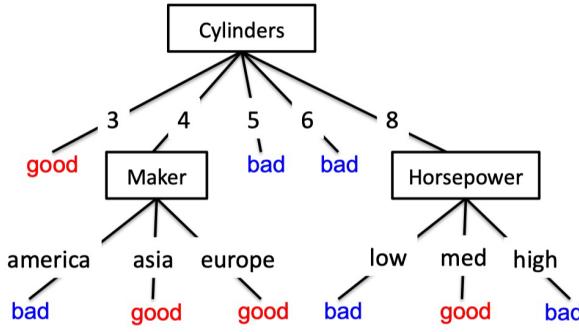


Figure 54: Decision tree structure. Each path from root to leaf defines a region through a sequence of threshold conditions. Note the interaction effects captured along branches—for example, combinations like “4 cylinders × continent” or “horsepower × low-med-high” encode how the effect of one variable depends on the value of another.

94.3 Prediction

Once a tree is constructed, prediction is straightforward: route the test point from root to leaf by evaluating splits, then return the leaf’s prediction.

Decision Tree Prediction

A tree with J leaves partitions feature space into disjoint regions R_1, \dots, R_J .

For a test point \tilde{x} :

$$f(\tilde{x}) = \sum_{j=1}^J c_j \cdot \mathbf{1}[\tilde{x} \in R_j]$$

where the leaf prediction c_j is:

- **Regression:** $c_j = \bar{y}_j = \frac{1}{|R_j|} \sum_{x_i \in R_j} y_i$ (mean of training responses in region j)
- **Classification:** $c_j = \arg \max_k \sum_{x_i \in R_j} \mathbf{1}[y_i = k]$ (majority vote among training labels in region j)

For probabilistic classification, we can return class proportions:

$$\hat{p}(Y = k \mid \tilde{x} \in R_j) = \frac{1}{|R_j|} \sum_{x_i \in R_j} \mathbf{1}[y_i = k]$$

What this formula means: The expression $f(\tilde{x}) = \sum_{j=1}^J c_j \cdot \mathbf{1}[\tilde{x} \in R_j]$ looks more complex than it is. The indicator function $\mathbf{1}[\tilde{x} \in R_j]$ equals 1 if \tilde{x} falls in region R_j and 0 otherwise. Since the regions are disjoint (non-overlapping) and exhaustive (covering all of feature space), exactly one indicator equals 1 and all others equal 0. So the sum simply returns the prediction c_j for whichever region contains \tilde{x} . This is just a mathematical way of saying “look up which region your test point falls into, and return that region’s prediction.”

Computational note: Prediction requires $O(\text{depth})$ comparisons per test point—typically $O(\log n)$ for balanced trees, making trees extremely fast at test time. This is far faster than methods like k -nearest neighbours (which require distance computations to all training points) or kernel methods.

94.4 Properties of Decision Trees

Advantages

- **Interpretability:** Visualise as flowchart; explain predictions as rule sequences (“If tenure < 2 years AND usage < 50 mins, predict churn”). Non-technical stakeholders can understand and verify the logic.
- **Flexibility:** Handle any input type (continuous, categorical, ordinal, mixed); capture nonlinear relationships and interactions automatically without explicit feature engineering
- **Non-parametric:** No assumptions about functional form or error distribution—the tree learns whatever structure is present in the data
- **Robust to outliers:** A single outlier only affects its own leaf’s prediction; the tree’s overall structure remains stable
- **Handle missingness:** Can split on “is missing?” or use surrogate splits (alternative splits that approximate the primary split when values are missing)
- **No standardisation needed:** Splits are threshold-based, so feature scales don’t matter—a feature ranging from 0-1 is treated the same as one ranging from 0-1000
- **Automatic feature selection:** Irrelevant features are simply never split on (though this can be unreliable for correlated features)
- **Fast prediction:** $O(\text{depth})$ per test point

NB!

Disadvantages:

- **Overfitting:** Unconstrained trees grow until each leaf is pure, fitting noise perfectly. A tree with n leaves can achieve zero training error by memorising every observation.
- **High variance:** Small changes in training data lead to very different tree structures (see Figure 55). Adding, removing, or slightly modifying a few training points can cascade into completely different trees.
- **Greedy optimisation:** Cannot find globally optimal tree (NP-hard problem). The greedy algorithm makes locally optimal choices that may be globally suboptimal.
- **Suboptimal accuracy:** Single trees often underperform other methods. The simple hierarchical decision-making process may miss nuances in the data.
- **Axis-aligned splits:** Cannot efficiently represent diagonal boundaries (requires many splits to approximate $x_1 + x_2 = c$). A linear classifier represents this with a single hyperplane; a tree needs a staircase of splits.
- **Instability:** The greedy algorithm can make different early splits based on noise, cascading into completely different subtrees. Due to hierarchical dependence, small changes to features can also fundamentally alter the tree.

Decision trees are **high-variance learners**—ideal candidates for ensemble methods that average many trees to smooth out the variance.

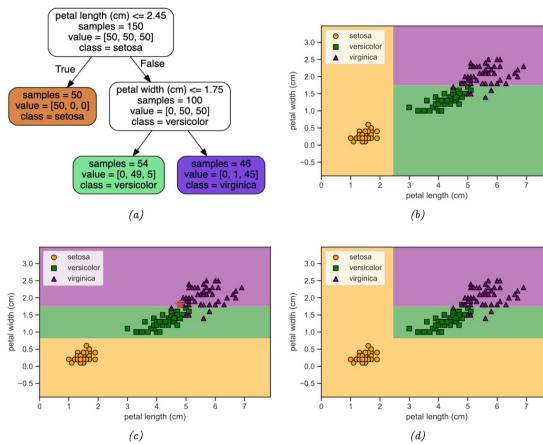


Figure 55: High variance: small changes in data lead to very different trees. Two bootstrap samples from the same population can yield dramatically different tree structures, even though they're estimating the same underlying relationship.

94.5 When to Use Trees

Decision Tree Use Cases

Use trees when:

- Interpretability is paramount (medical diagnosis, loan decisions, regulatory compliance)
- You have mixed feature types and don't want preprocessing
- You suspect complex interactions between features
- You need a quick baseline or exploratory analysis
- As base learners in ensembles (random forests, gradient boosting)

Consider alternatives when:

- You need smooth decision boundaries (use neural networks, RBF kernels)
- The true relationship is approximately linear (use linear/logistic regression)
- You need uncertainty quantification (use Bayesian methods, Gaussian processes)
- Maximum predictive accuracy is required (use ensembles or neural networks)

95 Tree Construction: Splitting Criteria

The core challenge in tree construction is choosing splits. At each node, we must select:

1. Which feature j to split on
2. What threshold t (or subset of categories) to use

We evaluate all possible (j, t) pairs and choose the one that best reduces “impurity”—a measure of how mixed the labels are within each resulting child node. The intuition is simple: we want each split to create child nodes that are more homogeneous than the parent. A “good” split separates the data into groups with distinct outcomes.

95.1 Splitting Continuous Features

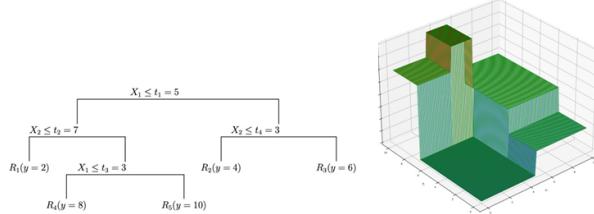


Figure 56: Threshold split: binary partition at threshold t . Prediction is constant within each region, creating the characteristic “stepped” shape of tree predictions.

For continuous feature x_j , a binary split at threshold t partitions the data:

$$\mathcal{D}_L = \{(x_i, y_i) : x_{ij} \leq t\}, \quad \mathcal{D}_R = \{(x_i, y_i) : x_{ij} > t\}$$

Candidate thresholds: Rather than searching over all real numbers (infinitely many), we only need to consider thresholds at the midpoints between consecutive sorted values of x_j . If x_j takes m unique values, there are $m - 1$ candidate thresholds. This is because any threshold between two consecutive values produces the same partition—what matters is which observations fall on each side, not the exact threshold value.

Why midpoints? Using midpoints $(v_i + v_{i+1})/2$ is conventional but not essential. Any value strictly between v_i and v_{i+1} produces the same split. Midpoints are aesthetically pleasing and provide some robustness to floating-point issues.

95.2 Splitting Categorical Features

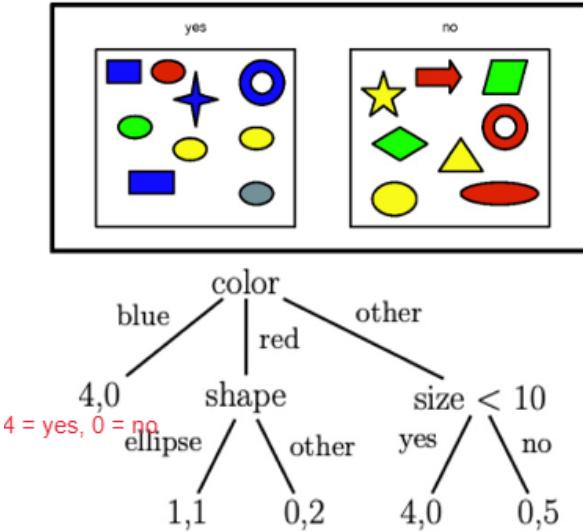


Figure 57: Categorical split: one branch per unique value.

For categorical feature x_j with k unique values:

Multi-way split: Create k branches, one per category. Simple but leads to overly complex trees with high-cardinality features. Each branch may contain few observations, making predictions unreliable.

Binary split: Partition categories into two groups S and S^c (where S^c denotes the complement—all categories not in S). For k categories, there are $2^{k-1} - 1$ possible partitions—exponential in k . This quickly becomes computationally prohibitive.

NB!

Computational shortcut for binary classification: When the target is binary, the optimal category partition can be found in $O(k \log k)$ by sorting categories by their class-1 proportion and searching over the $k - 1$ contiguous partitions. This follows from the theorem that the optimal split respects the ordering by conditional probability.

Why this works: Intuitively, if we want to separate classes, we should group categories with similar class distributions together. Sorting by class proportion puts “similar” categories adjacent, and the optimal partition is always a contiguous chunk of this sorted list.

For multi-class or regression: No such shortcut exists; we must either use heuristics, limit to binary splits, or accept exponential search time.

95.3 MSE Reduction for Regression

For regression trees, we seek splits that minimise squared error within each child node. The goal is to create child nodes where the responses y_i are clustered tightly around their mean.

Mean Squared Error Criterion

For a node with dataset \mathcal{D} , the MSE is:

$$\text{MSE}(\mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{(x_i, y_i) \in \mathcal{D}} (y_i - \bar{y})^2$$

where $\bar{y} = \frac{1}{|\mathcal{D}|} \sum_{(x_i, y_i) \in \mathcal{D}} y_i$ is the mean response.

A split into \mathcal{D}_L and \mathcal{D}_R has weighted MSE:

$$\text{MSE}_{\text{split}} = \frac{|\mathcal{D}_L|}{|\mathcal{D}|} \cdot \text{MSE}(\mathcal{D}_L) + \frac{|\mathcal{D}_R|}{|\mathcal{D}|} \cdot \text{MSE}(\mathcal{D}_R)$$

Choose the split (j, t) minimising $\text{MSE}_{\text{split}}$.

Equivalently, maximise the **variance reduction**:

$$\Delta \text{MSE} = \text{MSE}(\mathcal{D}) - \text{MSE}_{\text{split}} = \frac{|\mathcal{D}_L||\mathcal{D}_R|}{|\mathcal{D}|^2} (\bar{y}_L - \bar{y}_R)^2$$

This shows that good splits separate the means of the two child nodes.

Unpacking the weighted MSE: The weights $\frac{|\mathcal{D}_L|}{|\mathcal{D}|}$ and $\frac{|\mathcal{D}_R|}{|\mathcal{D}|}$ account for the relative sizes of the child nodes. A split creating one tiny node and one large node is penalised appropriately—the large node's MSE dominates the weighted average. This prevents gaming the criterion by creating tiny pure nodes at the expense of leaving most data poorly fit.

The variance reduction formula: The final expression $\frac{|\mathcal{D}_L||\mathcal{D}_R|}{|\mathcal{D}|^2} (\bar{y}_L - \bar{y}_R)^2$ reveals the geometry of good splits. The term $(\bar{y}_L - \bar{y}_R)^2$ measures how different the child means are—bigger separation is better. The term $\frac{|\mathcal{D}_L||\mathcal{D}_R|}{|\mathcal{D}|^2}$ is maximised when the split is balanced ($|\mathcal{D}_L| = |\mathcal{D}_R|$) and approaches zero for highly imbalanced splits. Together, they favour splits that create two reasonably-sized groups with different mean outcomes.

Derivation of Variance Reduction Formula

The derivation follows from expanding the MSE:

$$\begin{aligned}\text{MSE}(\mathcal{D}) &= \frac{1}{n} \sum_i (y_i - \bar{y})^2 = \frac{1}{n} \sum_i y_i^2 - \bar{y}^2 \\ \text{MSE}_{\text{split}} &= \frac{n_L}{n} \left(\frac{1}{n_L} \sum_{i \in L} y_i^2 - \bar{y}_L^2 \right) + \frac{n_R}{n} \left(\frac{1}{n_R} \sum_{i \in R} y_i^2 - \bar{y}_R^2 \right) \\ &= \frac{1}{n} \sum_i y_i^2 - \frac{n_L}{n} \bar{y}_L^2 - \frac{n_R}{n} \bar{y}_R^2\end{aligned}$$

Subtracting and using $\bar{y} = \frac{n_L}{n} \bar{y}_L + \frac{n_R}{n} \bar{y}_R$ yields the variance reduction formula.

Key step: The $\sum_i y_i^2$ terms cancel, leaving only terms involving the squared means. After algebra using the constraint that \bar{y} is a weighted average of \bar{y}_L and \bar{y}_R :

$$\Delta \text{MSE} = \frac{n_L}{n} \bar{y}_L^2 + \frac{n_R}{n} \bar{y}_R^2 - \bar{y}^2 = \frac{n_L n_R}{n^2} (\bar{y}_L - \bar{y}_R)^2$$

95.4 Gini Impurity for Classification

For classification with K classes, we need a measure of node “impurity”—how mixed the class labels are. A pure node contains only one class; an impure node contains a mixture.

Gini Impurity

For a node with class proportions p_1, \dots, p_K (where $p_k = \frac{\#\text{class } k}{n}$), the **Gini impurity** is:

$$G = \sum_{k=1}^K p_k (1 - p_k) = 1 - \sum_{k=1}^K p_k^2$$

Interpretation: G is the probability that a randomly chosen element, if labelled according to the class distribution, would be misclassified. Equivalently, it measures expected disagreement between two independent samples from the distribution.

Properties:

- $G = 0$ when node is pure (all samples from one class)
- G is maximised when all classes are equally likely: $G_{\max} = 1 - 1/K$
- For binary classification: $G = 2p(1 - p)$, maximised at $p = 0.5$ where $G = 0.5$

Understanding Gini impurity: The two equivalent forms $\sum_k p_k (1 - p_k)$ and $1 - \sum_k p_k^2$ give different intuitions:

- $\sum_k p_k (1 - p_k)$: For each class k , p_k is the probability of selecting that class, and $(1 - p_k)$ is the probability of misclassifying it (assigning any other class). Summing over classes gives the overall misclassification probability.
- $1 - \sum_k p_k^2$: The term $\sum_k p_k^2$ is the probability that two randomly drawn samples (with replacement) belong to the same class. Subtracting from 1 gives the probability they belong to different classes—a measure of “disagreement” or “diversity” in the node.

Why “Gini”? The impurity measure is related to the Gini coefficient from economics (measuring inequality), though the connection is historical rather than mathematical. Corrado Gini developed various measures of statistical dispersion in the early 20th century.

Derivation: Consider randomly selecting two samples with replacement from a node with class proportions (p_1, \dots, p_K) . The probability they have the same class is $\sum_k p_k^2$. The probability they differ is $1 - \sum_k p_k^2 = G$. Minimising Gini impurity maximises class homogeneity.

95.5 Entropy and Information Gain

An alternative to Gini impurity, rooted in information theory:

Entropy and Information Gain

The **entropy** of a node with class proportions p_1, \dots, p_K is:

$$H = - \sum_{k=1}^K p_k \log_2 p_k$$

where we take $0 \log 0 = 0$ by convention (justified by $\lim_{p \rightarrow 0^+} p \log p = 0$).

Interpretation: H measures the expected number of bits needed to encode a class label drawn from this distribution. Low entropy means the distribution is concentrated (predictable); high entropy means it's spread out (uncertain).

Properties:

- $H = 0$ when node is pure (no uncertainty—we know the class with certainty)
- H is maximised at uniform distribution: $H_{\max} = \log_2 K$
- For binary classification: $H = -p \log_2 p - (1-p) \log_2(1-p)$, maximised at $p = 0.5$ where $H = 1$ bit

The **information gain** from a split is:

$$\text{IG} = H(\mathcal{D}) - \left[\frac{|\mathcal{D}_L|}{|\mathcal{D}|} H(\mathcal{D}_L) + \frac{|\mathcal{D}_R|}{|\mathcal{D}|} H(\mathcal{D}_R) \right]$$

Choose the split maximising information gain.

The information theory perspective: Entropy comes from Claude Shannon's foundational work on communication. If you're transmitting class labels and can design an optimal coding scheme, the expected message length is exactly H bits. A pure node has $H = 0$: no message needed since the class is certain. A maximally uncertain node (uniform over K classes) requires $\log_2 K$ bits on average.

Information gain as mutual information: Information gain equals the mutual information between the split variable (which child node you fall into) and the class label, conditioned on reaching this node. In other words, knowing the split outcome gives you exactly IG bits of information about the class.

Why logarithms? The \log_2 makes bits the unit; using natural logarithm gives “nats.” The choice doesn't affect which split is optimal since $\log_a x = \log_b b \cdot \log_b x$ (just a constant factor).

95.6 Comparing Gini and Entropy

Gini vs Entropy

	Gini	Entropy
Range (binary)	$[0, 0.5]$	$[0, 1]$
At $p = 0.5$	0.5	1
Computation	Slightly faster	Requires log
Sensitivity	Less sensitive to p changes near 0.5	More sensitive

In practice, Gini and entropy produce very similar trees. The default in scikit-learn is Gini; information gain (entropy) is more common in the literature and has deeper theoretical connections to information theory and coding.

Both are concave functions of the class proportions, ensuring that any split reducing the weighted average impurity is beneficial. The concavity is important: it guarantees that splitting always helps (or at worst doesn't hurt) in terms of weighted impurity.

Mathematical Comparison

For binary classification with proportion $p \in [0, 1]$:

$$G(p) = 2p(1 - p)$$

$$H(p) = -p \log_2 p - (1 - p) \log_2(1 - p)$$

Taylor expansion around $p = 0.5$:

$$G(p) \approx 0.5 - 2(p - 0.5)^2$$

$$H(p) \approx 1 - \frac{2}{\ln 2}(p - 0.5)^2 \approx 1 - 2.885(p - 0.5)^2$$

Entropy has higher curvature at the maximum, making it slightly more aggressive at separating classes near the uniform distribution.

Practical implications: The higher curvature of entropy means it penalises “almost pure” nodes less than “moderately impure” nodes, relative to Gini. Near $p = 0$ or $p = 1$ (pure nodes), both measures are similarly flat. Near $p = 0.5$ (maximum impurity), entropy curves more sharply. In practice, this rarely matters—the two criteria select the same or very similar splits on most real datasets.

96 The Greedy Algorithm

NB!

Finding the globally optimal tree is **NP-hard**. The space of possible trees is exponential in the number of features and thresholds—there is no efficient algorithm guaranteed to find the tree minimising test error.

We must use **greedy** algorithms that make locally optimal decisions at each node, accepting that the resulting tree may be globally suboptimal. The greedy approach is the foundation of CART (Classification and Regression Trees) and essentially all practical tree implementations.

96.1 The Non-Differentiability Challenge

Unlike neural networks or linear models, decision trees cannot be optimised using gradient-based methods. The reason is fundamental: the threshold parameters $\theta = \{(j_1, t_1), (j_2, t_2), \dots\}$ that define splits create **discontinuous** changes in predictions.

When an observation moves from one side of a threshold to another, its prediction can jump discontinuously. There is no smooth gradient to follow. Consider moving threshold t slightly: some observations suddenly change which leaf they fall into, causing their predictions to jump. The loss function is piecewise constant in the thresholds, with discontinuities at each training point's feature value.

This rules out standard optimisation techniques like gradient descent, stochastic gradient descent, or any method relying on derivatives.

96.2 Recursive Splitting Algorithm

CART: Classification and Regression Trees

Input: Dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$, stopping criteria

BuildTree(\mathcal{D}):

1. If stopping criterion met: return leaf with prediction $c = \bar{y}$ (regression) or $c = \text{mode}(y)$ (classification)
2. For each feature j and threshold t :
 - Compute split quality: $\text{Score}(j, t) = \text{ImpurityReduction}(\mathcal{D}, j, t)$
3. Select best split: $(j^*, t^*) = \arg \max_{j,t} \text{Score}(j, t)$
4. Partition: $\mathcal{D}_L = \{(x, y) : x_j^* \leq t^*\}$, $\mathcal{D}_R = \{(x, y) : x_j^* > t^*\}$
5. Return internal node with:
 - Split rule: $x_{j^*} \leq t^*$
 - Left child: **BuildTree(\mathcal{D}_L)**
 - Right child: **BuildTree(\mathcal{D}_R)**

Walking through the algorithm: Starting with all data at the root, we exhaustively evaluate every possible split (every feature j , every threshold t) and pick the one that most reduces impurity. We then partition the data according to this split and recursively apply the same

procedure to each child. The recursion terminates when a stopping condition is met (e.g., too few samples, maximum depth reached, or no split improves impurity).

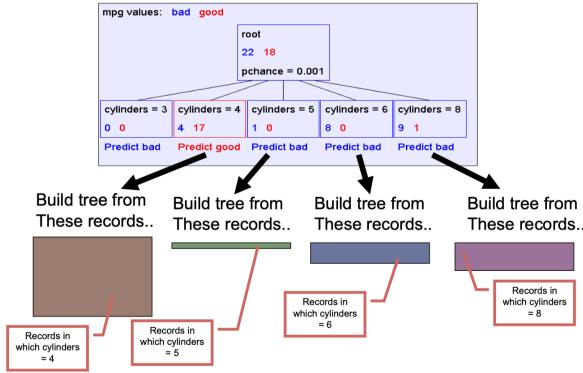


Figure 58: Greedy recursive splitting: at each node, choose the locally best split without considering downstream consequences. This myopic approach is computationally efficient but may miss globally better tree structures.

Time complexity: For n samples and p features, each split requires $O(np)$ operations (evaluate all features and thresholds). A balanced tree has $O(\log n)$ levels, giving total complexity $O(n p \log n)$. An unbalanced tree can have $O(n)$ levels, giving $O(n^2 p)$ worst case. Sorting features once and updating incrementally can improve constants.

96.3 Stopping Criteria

Without constraints, a tree will grow until every leaf contains a single training point (or all points in a leaf have identical features). This perfectly fits training data but badly overfits.

Common Stopping Criteria (Pre-Pruning)

- **Maximum depth:** Stop when path length from root reaches limit
- **Minimum samples per leaf:** Don't create leaves smaller than threshold
- **Minimum samples to split:** Don't split nodes smaller than threshold
- **Minimum impurity decrease:** Don't split if impurity reduction is below threshold
- **Maximum leaf nodes:** Stop when tree has enough leaves

These are **hyperparameters**—tune via cross-validation.

NB!

The stopping dilemma: Strict stopping criteria may prevent useful splits. Consider XOR: no single split on x_1 or x_2 reduces impurity, but the combination of two splits perfectly separates the classes. Early stopping would miss this.

Example: In XOR, points at $(0, 0)$ and $(1, 1)$ belong to class 0; points at $(0, 1)$ and $(1, 0)$ belong to class 1. Splitting on x_1 alone gives each child a 50-50 class mix—no improvement. Same for x_2 . But splitting on x_1 first, then x_2 within each child, achieves perfect separation.

This motivates **post-pruning**: grow a full tree, then prune back.

97 Pruning

97.1 Why Trees Overfit

An unconstrained tree will grow until every leaf is pure (or contains identical feature vectors). This happens because:

1. The greedy algorithm always finds *some* split that reduces training impurity (even if by a tiny amount due to noise)
2. There's no penalty for adding nodes—more leaves always means equal or lower training error
3. Training error monotonically decreases as the tree grows

A fully-grown tree with n leaves (one per training point) achieves zero training error but captures every quirk and noise pattern in the data. This is memorisation, not learning. Test error initially decreases as the tree captures real structure, then increases as it fits noise—the classic bias-variance trade-off.

97.2 Pre-Pruning vs Post-Pruning

Pruning Strategies

Pre-pruning (early stopping):

- Apply stopping criteria during tree construction
- Fast: never builds subtrees that would be pruned
- Risk: may stop too early (XOR problem), miss beneficial deep splits
- Hyperparameters: max depth, min samples per leaf, min impurity decrease

Post-pruning (grow then prune):

- Grow full tree (or nearly full, with minimal stopping criteria)
- Prune back nodes that don't improve validation error
- More computation but often better results
- Can discover beneficial deep splits that pre-pruning would miss

The philosophical difference: Pre-pruning says “don’t build complexity you don’t need.” Post-pruning says “build everything, then remove what doesn’t help.” The latter is more conservative—it won’t miss useful structure—but requires more computation.

97.3 Cost-Complexity Pruning (CART)

The most principled approach to post-pruning is **cost-complexity pruning** (also called *weakest link pruning*), used in CART.

Cost-Complexity Criterion

For a tree T , define the **cost-complexity** objective:

$$R_\alpha(T) = R(T) + \alpha|T|$$

where:

- $R(T)$ = training error (misclassification rate or MSE)
- $|T|$ = number of leaf nodes (tree size)
- $\alpha \geq 0$ = complexity parameter (penalty per leaf)

For each α , find the subtree $T_\alpha \subseteq T_{\max}$ minimising $R_\alpha(T)$.

Key insight: As α increases, the optimal subtree shrinks (fewer leaves). There exists a nested sequence of subtrees:

$$T_{\max} = T_0 \supseteq T_1 \supseteq T_2 \supseteq \cdots \supseteq T_K = \{\text{root}\}$$

where T_{k+1} is obtained by pruning the “weakest link” from T_k —the internal node whose removal increases $R(T)$ least per leaf removed.

Understanding the cost-complexity objective: The term $R(T)$ rewards accuracy; the term $\alpha|T|$ penalises complexity. With $\alpha = 0$, there’s no penalty for complexity, so the full tree is optimal. As α increases, each additional leaf must “pay for itself” by reducing training error by at least α . Large α favours small trees; small α favours large trees.

This is directly analogous to LASSO regularisation in linear models, where the penalty term balances fit against complexity. The parameter α plays the same role as the regularisation parameter λ .

Weakest Link Pruning Algorithm

For each internal node t in tree T , compute:

$$g(t) = \frac{R(t) - R(T_t)}{|T_t| - 1}$$

where $R(t)$ is the error if we replace subtree T_t with a leaf, and $|T_t|$ is the number of leaves in subtree T_t .

Interpretation: $g(t)$ is the increase in training error per leaf pruned. The node with smallest $g(t)$ is the “weakest link”—it contributes least to accuracy per unit complexity.

Algorithm:

1. Start with $T_0 = T_{\max}$
2. Find weakest link: $t^* = \arg \min_t g(t)$
3. Prune: $T_{k+1} = T_k$ with subtree at t^* replaced by leaf
4. Record $\alpha_k = g(t^*)$
5. Repeat until only root remains

This produces the sequence T_0, T_1, \dots, T_K and corresponding $\alpha_0 < \alpha_1 < \dots < \alpha_K$.

Why “weakest link”? The metaphor is apt: we identify the part of the tree that contributes least to performance (relative to its complexity cost) and remove it. The node with smallest $g(t)$ is the one where pruning costs the least per leaf removed. Iteratively removing weakest links gives a sequence of trees of decreasing complexity.

Efficiency note: The algorithm produces the entire sequence of optimally pruned trees in a single pass through the full tree. We don’t need to solve a separate optimisation problem for each α —the nested sequence covers all possible optimal trees.

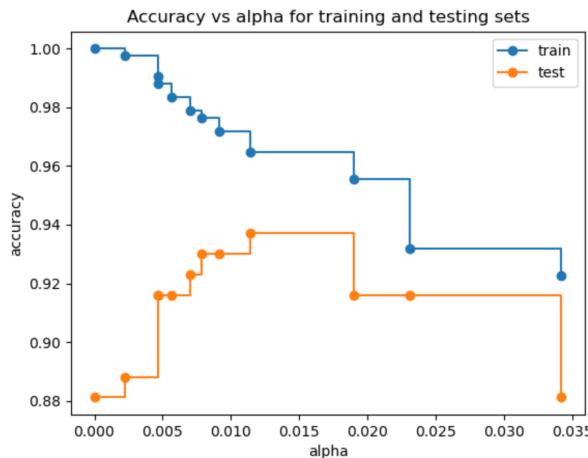


Figure 59: Cost-complexity pruning: test error improves as we prune back the overfitted tree. The optimal α is selected by cross-validation.

Selecting α : Use cross-validation. For each fold:

1. Grow full tree on training portion

2. Generate pruning sequence
3. Evaluate each T_α on validation portion

Average validation errors across folds; select α with minimum error (or use one-standard-error rule from Week 4: choose the simplest model within one standard error of the minimum).

98 Worked Example: Building a Classification Tree

Let's construct a classification tree step-by-step on a small dataset.

Example Dataset

Predicting whether customers buy a product based on Age and Income:

Age	Income (\$k)	Buys?
25	40	No
35	60	No
45	80	Yes
20	25	No
35	120	Yes
52	90	Yes
23	35	No
40	100	Yes

Classes: 4 Yes, 4 No. Initial proportions: $p_{\text{Yes}} = 0.5$, $p_{\text{No}} = 0.5$.

Step 1: Compute initial impurity

Gini impurity at root:

$$G_{\text{root}} = 2 \times 0.5 \times 0.5 = 0.5$$

This is the maximum possible Gini impurity for binary classification—the classes are perfectly balanced.

Step 2: Evaluate all possible splits

Candidate thresholds for Age: Sort unique values: 20, 23, 25, 35, 40, 45, 52. Midpoints: 21.5, 24, 30, 37.5, 42.5, 48.5 (note: 35 appears twice, so midpoint between 25 and 35 is 30).

Candidate thresholds for Income: Sort unique values: 25, 35, 40, 60, 80, 90, 100, 120. Midpoints: 30, 37.5, 50, 70, 85, 95, 110.

Let's evaluate Income ≤ 50 :

- Left (≤ 50): Observations with Income $\in \{40, 25, 35\}$ plus one with 60... wait, let me recount.
- Observations with Income ≤ 50 : (25, 40, No), (20, 25, No), (23, 35, No). That's 3 samples, but we need 4 in left.
- Actually Income = 40 ≤ 50 : Yes. Income = 60 > 50 . So left has Income $\in \{40, 25, 35\}$: 3 samples.

Let me recalculate more carefully. Looking at all incomes: 40, 60, 80, 25, 120, 90, 35, 100.

For threshold $t = 50$:

- Left (≤ 50): Income $\in \{40, 25, 35\}$, corresponding to observations 1, 4, 7. Labels: No, No, No. That's 3 samples, all No.

- Right (> 50): Income $\in \{60, 80, 120, 90, 100\}$, corresponding to observations 2, 3, 5, 6, 8. Labels: No, Yes, Yes, Yes, Yes. That's 5 samples: 1 No, 4 Yes.

Gini for left ($n_L = 3$): $G_L = 2 \times 1 \times 0 = 0$ (pure—all No)

Gini for right ($n_R = 5$): $p_{\text{Yes}} = 4/5 = 0.8$, so $G_R = 2 \times 0.8 \times 0.2 = 0.32$

Weighted Gini after split:

$$G_{\text{split}} = \frac{3}{8} \times 0 + \frac{5}{8} \times 0.32 = 0.20$$

Gini reduction: $\Delta G = 0.5 - 0.20 = 0.30$

Let me try another threshold. Income ≤ 70 :

- Left (≤ 70): Income $\in \{40, 60, 25, 35\}$, observations 1, 2, 4, 7. Labels: No, No, No, No. 4 samples, all No. $G_L = 0$.
- Right (> 70): Income $\in \{80, 120, 90, 100\}$, observations 3, 5, 6, 8. Labels: Yes, Yes, Yes, Yes. 4 samples, all Yes. $G_R = 0$.

Weighted Gini:

$$G_{\text{split}} = \frac{4}{8} \times 0 + \frac{4}{8} \times 0 = 0$$

Gini reduction: $\Delta G = 0.5 - 0 = 0.5$ (maximum possible!)

This split perfectly separates the classes. Let's verify by checking an Age split.

Try Age ≤ 30 :

- Left (≤ 30): Ages $\in \{25, 20, 23\}$, observations 1, 4, 7. Labels: No, No, No. 3 samples. $G_L = 0$.
- Right (> 30): Ages $\in \{35, 45, 35, 52, 40\}$, observations 2, 3, 5, 6, 8. Labels: No, Yes, Yes, Yes, Yes. 5 samples: 1 No, 4 Yes.
 - $p_{\text{Yes}} = 4/5 = 0.8$
 - $G_R = 2 \times 0.8 \times 0.2 = 0.32$

Weighted Gini:

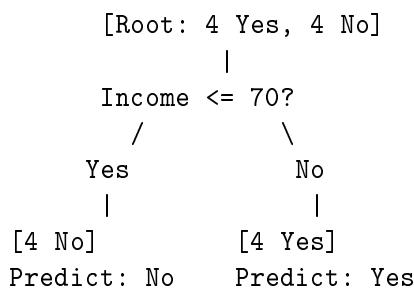
$$G_{\text{split}} = \frac{3}{8} \times 0 + \frac{5}{8} \times 0.32 = 0.20$$

Gini reduction: $\Delta G = 0.5 - 0.20 = 0.30$

Income ≤ 70 is better ($\Delta G = 0.5 > 0.20$).

Step 3: Apply best split

Since Income ≤ 70 achieves perfect separation, the tree is complete:



Worked Example Summary

Key observations:

- A single split on Income perfectly separates the classes
- We evaluated all candidate splits and chose the one maximising Gini reduction
- The greedy algorithm found the optimal tree for this dataset
- With noisy data, we'd likely need multiple splits and pruning
- The threshold $t = 70$ (midpoint between 60 and 80) works; any value in $(60, 80)$ would produce the same split

99 Trees as Piecewise Constant Approximations

99.1 The Approximation View

A regression tree with J leaves is fundamentally a **piecewise constant** function:

$$f(x) = \sum_{j=1}^J c_j \cdot \mathbf{1}[x \in R_j]$$

where R_1, \dots, R_J partition the feature space into axis-aligned rectangles.

What this means geometrically: Imagine the feature space as a floor, and the prediction as a height. A tree creates a “staircase” surface where the floor is divided into rectangular tiles, and each tile is at a constant height. The surface is flat within each tile but can jump discontinuously between tiles.

Comparison with Other Approximations

Linear models: $f(x) = \beta_0 + \sum_j \beta_j x_j$

- Single hyperplane; global assumption about relationship
- Cannot capture interactions without explicit feature engineering (adding $x_1 \cdot x_2$ terms)
- Extrapolates linearly beyond training data

Polynomial models: $f(x) = \sum_{|\alpha| \leq d} \beta_\alpha x^\alpha$

- Smooth approximation to any function (Weierstrass theorem)
- Degree d controls flexibility; oscillation issues at boundaries
- Can capture nonlinear relationships and interactions

Trees (piecewise constant): $f(x) = \sum_j c_j \cdot \mathbf{1}[x \in R_j]$

- Discontinuous step function
- Adapts complexity locally (more splits where needed)
- Axis-aligned regions; cannot efficiently represent diagonal boundaries

Key insight: Trees are like histograms with adaptive bin boundaries—simple within each region, complex through the number and placement of regions.

99.2 Comparison with Linear Methods

Trees vs Linear Models

	Linear	Tree
Decision boundary	Hyperplane	Axis-aligned rectangles
Interactions	Must be specified	Automatic
Extrapolation	Linear	Constant (last leaf)
Interpretability	Coefficients	Rules
Variance	Low	High
Bias (if linear truth)	Low	High
Bias (if complex truth)	High	Low

Rule of thumb: If the true relationship is approximately linear, use linear models. If relationships are complex with interactions, trees (especially ensembles) often win.

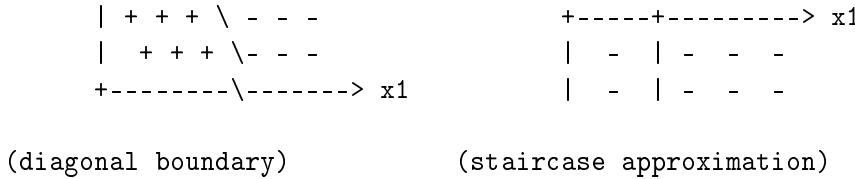
Consider the decision boundaries:

Linear classifier:

$$\begin{array}{ccccccc} & x_2 \\ & ^\wedge \\ | & + & + & + & + \\ | & + & + & + & + \end{array}$$

Tree classifier:

$$\begin{array}{ccccccc} & x_2 \\ & ^\wedge \\ | & - & | & + & + & + \\ | & - & | & + & + & + \end{array}$$



A linear classifier can represent diagonal boundaries with a single hyperplane. A tree must approximate this with many axis-aligned splits, creating a “staircase” boundary. This is the fundamental limitation of axis-aligned splits: boundaries that don’t align with the feature axes require many splits to approximate.

Quantifying the inefficiency: To approximate a diagonal line $x_1 + x_2 = c$ with accuracy ϵ in a unit square, a tree needs $O(1/\epsilon)$ leaves. This is the “staircase approximation” cost of axis-aligned splits.

100 Ensemble Methods: Reducing Variance

The high variance of decision trees makes them ideal candidates for **ensemble methods**—combining many models to reduce variance while preserving flexibility.

100.1 The Bias-Variance Motivation

Recall from Week 3 that test error decomposes as:

$$\mathbb{E}[(Y - \hat{f}(X))^2] = \underbrace{\text{Var}(\hat{f}(X))}_{\text{reducible}} + \underbrace{\text{Bias}^2(\hat{f}(X))}_{\text{irreducible}} + \sigma^2$$

Decision trees have:

- **Low bias:** Deep trees can approximate complex functions—they’re flexible enough to capture almost any pattern
- **High variance:** Small data changes \Rightarrow different tree structure—different training sets give very different models

If we could reduce variance without increasing bias, we’d get better predictions. Ensemble methods achieve this by averaging many high-variance, low-bias models.

Variance Reduction via Averaging

Consider M independent estimates $\hat{f}_1, \dots, \hat{f}_M$, each with variance σ^2 and the same bias.

The averaged estimate $\bar{f} = \frac{1}{M} \sum_{m=1}^M \hat{f}_m$ has:

- Same bias as individual estimates (averaging doesn’t change expectation)
- Variance $\frac{\sigma^2}{M}$ (reduced by factor M)

The catch: Our trees aren’t independent—they’re trained on the same data. But we can make them *approximately* independent through resampling.

The key insight: Averaging reduces variance when the estimators have uncorrelated (or weakly correlated) errors. Even if individual trees make mistakes, they’ll make *different* mistakes, which cancel out when averaged. The more decorrelated the trees, the greater the variance reduction.

100.2 Bagging (Bootstrap Aggregating)

Bagging turns the instability of decision trees from a weakness into a strength. By training trees on different bootstrap samples, we create diverse models whose averaged predictions are more stable than any individual.

Bagging Algorithm

1. Create M bootstrap samples: sample n points **with replacement** from training data
2. Fit an unpruned decision tree to each bootstrap sample
3. Aggregate predictions:
 - Regression: $\hat{f}(x) = \frac{1}{M} \sum_{m=1}^M \hat{f}_m(x)$
 - Classification: $\hat{y}(x) = \text{majority vote of } \hat{f}_1(x), \dots, \hat{f}_M(x)$

Each bootstrap sample contains approximately 63.2% unique observations (on average). The remaining $\approx 36.8\%$ are **out-of-bag (OOB)**—not used to train that tree.

Why bootstrap sampling? We want to simulate having multiple independent training sets, but we only have one. Bootstrap sampling creates “pseudo-datasets” by resampling with replacement. Each bootstrap sample is like a slightly different version of reality—some observations appear multiple times, others don’t appear at all. This variation is what makes the trees different.

Why 63.2%? The probability that observation i is *not* selected in any of n draws with replacement is $(1 - 1/n)^n \rightarrow e^{-1} \approx 0.368$ as $n \rightarrow \infty$. So approximately $1 - 0.368 = 0.632$ or 63.2% of observations appear at least once in each bootstrap sample.

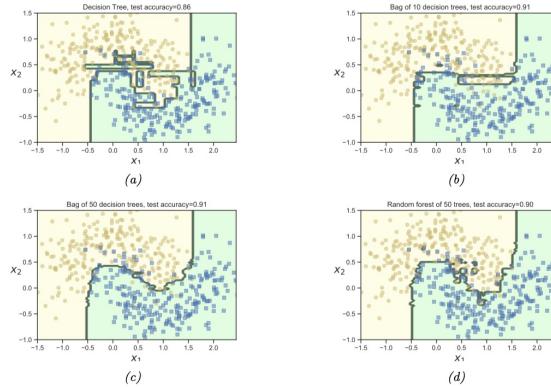


Figure 60: Bagging smooths out the jagged predictions of individual trees. (a) A single tree shows high variance with overfit predictions. As we aggregate more trees (b–d), the ensemble prediction smooths out, reducing variance while maintaining the ability to capture the underlying signal.

Why Bagging Works

- **Variance reduction:** Each tree sees different training data, making different splits. Averaging reduces idiosyncratic variance.
- **Bias preserved:** Each tree is grown deep (low bias); average of low-bias estimators is low-bias.
- **OOB error:** Use each tree's OOB samples for validation—free error estimate without holdout set!
- **Parallelisable:** Trees are independent; train on separate cores/machines.

Key insight: Averaging many high-variance models produces a low-variance ensemble—if the models' errors are not too correlated.

100.2.1 Out-of-Bag (OOB) Error Estimation

The $\sim 37\%$ of observations not in each bootstrap sample provide “free” validation:

OOB Error Estimation

For each observation x_i :

1. Identify all trees where x_i was *not* in the bootstrap sample (i.e., x_i was OOB)
2. Average predictions from only those trees: $\hat{f}_{\text{OOB}}(x_i) = \frac{1}{|\mathcal{T}_i|} \sum_{m \in \mathcal{T}_i} \hat{f}_m(x_i)$
3. Compare to true y_i

The OOB error is computed across all observations, using only trees for which each observation was held out. This provides an honest estimate of generalisation error—similar to cross-validation but without the computational cost of re-fitting.

Computational advantage: OOB error estimation comes “for free”—we don't need to set aside a validation set or perform cross-validation. Each observation serves as a test case for approximately $M \times 0.368$ trees.

100.2.2 Variance of Correlated Estimators

The effectiveness of bagging depends on how correlated the trees are:

Variance of Correlated Estimators

If M estimators have variance σ^2 and pairwise correlation ρ , the variance of their average is:

$$\text{Var} \left(\frac{1}{M} \sum_{m=1}^M \hat{f}_m \right) = \frac{\sigma^2}{M} + \frac{M-1}{M} \rho \sigma^2 = \rho \sigma^2 + \frac{1-\rho}{M} \sigma^2$$

As $M \rightarrow \infty$, variance approaches $\rho \sigma^2$, not zero. High correlation limits variance reduction.

Unpacking the formula: The variance has two terms:

- $\frac{(1-\rho)\sigma^2}{M}$: This term shrinks as M increases—the benefit of averaging

- $\rho\sigma^2$: This term is a floor that doesn't decrease with M —the cost of correlation

If $\rho = 0$ (independent trees), variance drops to σ^2/M —perfect! If $\rho = 1$ (identical trees), variance stays at σ^2 —no benefit from averaging. Real bagged trees have $0 < \rho < 1$, so we get some but not unlimited benefit from averaging.

100.2.3 Variance Estimation with Bagging

Beyond prediction, bagging provides a natural way to estimate the **uncertainty** in predictions:

Variance of Predictions

For a test point x , the variance across the M tree predictions provides an estimate of prediction uncertainty:

$$\widehat{\text{Var}}[\hat{f}(x)] = \frac{1}{M-1} \sum_{m=1}^M \left(\hat{f}_m(x) - \bar{f}(x) \right)^2$$

where $\bar{f}(x) = \frac{1}{M} \sum_m \hat{f}_m(x)$ is the ensemble prediction.

This variance estimate is **not constant** across the feature space—it tends to be higher in regions with:

- Fewer training observations (less information)
- Steeper prediction surfaces (small changes in data have larger effects)
- Greater disagreement among individual trees

The **infinitesimal jackknife** and related techniques can provide more refined variance estimates by analysing how predictions change when individual observations are perturbed.

Practical use: This variance estimate gives confidence intervals for predictions. If trees strongly disagree on a prediction, the variance is high, signalling uncertainty. If they agree, variance is low, signalling confidence. This is valuable for decision-making under uncertainty.

100.3 Random Forests

Random forests add a second source of randomness to further decorrelate the trees.

Random Forest = Bagging + Feature Subsampling

At each split (not just each tree):

1. Randomly select m features from all p features
2. Find the best split using **only those m features**
3. Apply the split; recurse on children

Typical choices: $m \approx \sqrt{p}$ for classification, $m \approx p/3$ for regression.

This forces trees to use different features at each split, reducing correlation between trees.

Why feature subsampling at each split? In bagging, if one feature is overwhelmingly predictive, most trees will split on it first. This makes trees similar—they make correlated errors. By forcing trees to consider only a random subset of features at each split, we ensure trees develop different structures. Sometimes a tree won't have access to the “best” feature and must find an alternative, leading to diversity.

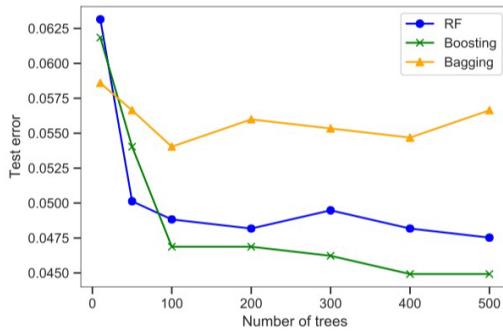


Figure 61: Random forest: feature subsampling decorrelates trees. More trees generally improve performance (with diminishing returns). The ensemble captures complex structure that individual trees miss.

Random Forest Properties

- **Individual trees are worse:** Feature restriction prevents always choosing the “best” split, increasing bias slightly
- **But ensemble is better:** Lower correlation between trees means greater variance reduction from averaging
- **Scales well:** More trees always help (diminishing returns after 100-500, but never hurts)
- **Few hyperparameters:** Main ones are M (number of trees) and m (features per split)
- **Remarkably robust:** Works well “out of the box” on many problems with minimal tuning

The Decorrelation Effect

Without feature subsampling, if one feature is strongly predictive, most trees will split on it first. This makes trees similar—**correlated**.

Correlated trees make correlated errors. When we average correlated predictions, variance reduction is limited (recall: variance approaches $\rho\sigma^2$ as $M \rightarrow \infty$).

Feature subsampling reduces ρ by preventing the same features from dominating every tree. This allows greater variance reduction through averaging—the $(1-\rho)/M$ term shrinks faster when ρ is smaller.

Trade-off: Each individual tree is worse (higher bias from feature restriction), but the ensemble is better (lower variance from decorrelation). The net effect is usually positive.

Random forests are, in some sense, an **atheoretical heuristic**—there's no deep theoretical justification for why feature subsampling works so well. The variance reduction formula above

provides some insight, but the precise optimal choice of m is empirical. Yet random forests are remarkably effective across a wide range of problems, often achieving near-state-of-the-art performance with minimal tuning.

NB!

Interpretability cost: Random forests sacrifice the interpretability of single trees. You cannot draw a simple flowchart or extract clean rules. The “reasoning” of the ensemble is opaque—no single path through a decision tree explains the prediction.

Feature importance measures (e.g., mean decrease in impurity, permutation importance) provide some insight but are not a substitute for the transparent logic of a single tree.

For interpretability, use a single tree (with pruning) or extract rules from the forest. For accuracy, use the full ensemble.

100.4 Preview: Boosting

Bagging and random forests reduce variance by training trees independently and averaging. **Boosting** takes a different approach: train trees **sequentially**, with each tree correcting errors from the previous ensemble.

Bagging vs Boosting Preview

	Bagging/RF	Boosting
Training	Parallel	Sequential
Trees	Independent	Corrective
Primary benefit	Reduce variance	Reduce bias
Tree depth	Deep (unpruned)	Shallow (“stumps”)
Overfitting risk	Low	Higher (needs regularisation)

Boosting is covered in detail in Week 8, including AdaBoost, gradient boosting, and XGBoost.

Intuition for why boosting is different: Bagging asks “what if we had different training data?” and averages the answers. Boosting asks “where are we still making mistakes?” and focuses subsequent models on those mistakes. Bagging is parallel and reduces variance; boosting is sequential and reduces bias.

101 Summary

Key Concepts: Decision Trees and Ensembles

1. **Decision trees:** Recursive binary partitioning; predict constant value per region
2. **Splitting criteria:** Gini impurity or entropy (classification), MSE reduction (regression)
3. **Greedy algorithm:** Locally optimal splits; globally suboptimal tree (NP-hard problem)
4. **High variance:** Small data changes \Rightarrow very different tree structure
5. **Pruning:** Cost-complexity criterion balances accuracy vs tree size; select α via CV
6. **Bagging:** Average bootstrap trees to reduce variance; OOB error for free validation
7. **Random forests:** Bagging + feature subsampling for tree decorrelation
8. **Trees as approximators:** Piecewise constant with adaptive, axis-aligned regions
9. **Ensemble principle:** Average high-variance, low-bias models \Rightarrow low-variance ensemble
10. **Variance estimation:** Disagreement across trees quantifies prediction uncertainty

Practical Recommendations

- **Single tree:** Use when interpretability is essential; apply cost-complexity pruning
- **Random forest:** Default choice for most tabular data problems; robust out-of-box
- **Number of trees:** Start with 100-500; monitor OOB error for diminishing returns
- **Feature subsampling:** $m = \sqrt{p}$ (classification) or $m = p/3$ (regression)
- **Hyperparameter tuning:** Focus on max depth, min samples per leaf, number of features
- **Uncertainty:** Use variance across trees to estimate prediction confidence

Chapter Summary

Boosting constructs strong classifiers by **sequentially** combining weak learners, with each iteration focusing on mistakes from previous rounds. Unlike bagging (parallel, variance reduction), boosting is **sequential** and primarily **reduces bias**. Key concepts: **AdaBoost** reweights misclassified samples and uses exponential loss; **gradient boosting** generalises this by fitting successive models to the **negative gradient** of any differentiable loss—equivalent to gradient descent in function space. **XGBoost** and **LightGBM** add regularisation and second-order optimisation for state-of-the-art performance on tabular data. Critical hyperparameters: number of iterations (use early stopping), learning rate (smaller = more iterations needed but better generalisation), and tree depth (shallow trees work best—stumps for AdaBoost, depth 3–6 for gradient boosting).

102 Review: Bagging vs Boosting

Before diving into boosting, let us contrast it with the bagging approach from Week 7. Both are ensemble methods—they combine multiple models to produce a single, more powerful predictor—but their strategies are fundamentally different.

Bagging vs Boosting: The Core Distinction

	Bagging / Random Forests	Boosting
Training	Parallel (independent)	Sequential (corrective)
Combination	Average / majority vote	Weighted sum
Base learners	Deep trees (low bias)	Shallow trees (high bias)
Primary benefit	Reduces variance	Reduces bias
Overfitting	Resistant (more trees $\not\Rightarrow$ overfit)	Prone (needs regularisation)

Bagging fits trees independently to bootstrap samples, then averages their predictions. Each tree is deep (low bias, high variance); averaging reduces variance while preserving low bias. Adding more trees never hurts—variance keeps decreasing (with diminishing returns), and bias stays constant. This is the “wisdom of crowds” approach: many independent opinions, averaged together, tend to be accurate.

Boosting fits trees sequentially, with each tree correcting errors from the current ensemble. Each tree is shallow (high bias, low variance); the sequential combination reduces bias. However, too many iterations can overfit—the ensemble eventually memorises training data. This is the “expert consultation” approach: each new expert focuses specifically on cases the previous experts got wrong.

The choice between bagging and boosting depends on your problem’s characteristics and your priorities:

When to Use Which

Use bagging/random forests when:

- You want robust out-of-the-box performance with minimal tuning
- Training time is limited (parallelisable across cores/machines)
- You're concerned about overfitting (random forests are naturally resistant)
- Interpretability is somewhat important (feature importance is reliable)

Use boosting when:

- Maximum predictive accuracy is the primary goal
- You have time and resources for hyperparameter tuning
- The problem requires fitting subtle patterns (boosting excels at reducing bias)
- You're working on a Kaggle competition (boosting dominates tabular data leaderboards)

In practice: Gradient boosting (XGBoost, LightGBM) often achieves the best performance on tabular data, but requires more careful tuning than random forests. Start with random forests as a baseline, then try boosting if you need to squeeze out additional performance.

103 Motivation: Correlated Errors in Ensembles

The fundamental challenge in ensemble learning, particularly with decision tree-based methods like Random Forests, is the **correlation of errors** among the individual trees.

Despite efforts to diversify the trees—through bootstrapping the data or manipulating input features—the errors made by individual trees can still be correlated. This correlation diminishes the ensemble's ability to reduce overall error through averaging. To understand why, we need to examine the mathematics of variance reduction in ensembles.

Variance of Correlated Estimators

If M estimators have variance σ^2 and pairwise correlation ρ , the variance of their average is:

$$\text{Var}\left(\frac{1}{M} \sum_{m=1}^M \hat{f}_m\right) = \rho\sigma^2 + \frac{1-\rho}{M}\sigma^2$$

What this formula tells us:

- The first term $\rho\sigma^2$ is the **irreducible variance**—it does not decrease as $M \rightarrow \infty$
- The second term $\frac{1-\rho}{M}\sigma^2$ **does** decrease with more trees
- As $M \rightarrow \infty$, variance approaches $\rho\sigma^2$, not zero

If trees make similar mistakes (ρ high), averaging provides limited benefit. The correlation ρ acts as a floor on how much variance reduction we can achieve.

To derive this formula, recall that for random variables X_1, \dots, X_M :

$$\text{Var}\left(\sum_m X_m\right) = \sum_m \text{Var}(X_m) + 2 \sum_{m < m'} \text{Cov}(X_m, X_{m'})$$

If each X_m has variance σ^2 and all pairs have covariance $\rho\sigma^2$ (where ρ is the correlation), then:

$$\begin{aligned} \text{Var}\left(\frac{1}{M} \sum_m X_m\right) &= \frac{1}{M^2} \left[M\sigma^2 + 2 \binom{M}{2} \rho\sigma^2 \right] \\ &= \frac{1}{M^2} [M\sigma^2 + M(M-1)\rho\sigma^2] \\ &= \frac{\sigma^2}{M} + \frac{(M-1)\rho\sigma^2}{M} \\ &= \frac{\sigma^2(1-\rho)}{M} + \rho\sigma^2 \end{aligned}$$

The ensemble's main strength is its ability to reduce the variance component of error by averaging out uncorrelated errors from diverse models. However, if errors are correlated, this variance reduction mechanism breaks down.

NB!

Despite bootstrapping and feature subsampling, tree errors in random forests can still be correlated—especially when one feature is much more predictive than others. When trees independently fit the same data patterns, they tend to make similar mistakes on the same observations. Boosting addresses this by **sequentially** fitting trees to correct previous errors, explicitly targeting the mistakes rather than hoping randomness decorrelates them.

103.1 Ensemble Prediction Function

We can formalise the general ensemble method as a prediction function. Both bagging and boosting can be written in this form, but they differ in how the components are trained and weighted.

Ensemble Prediction

$$f(x; \theta, w) = \sum_{m=1}^M w_m F_m(x; \theta)$$

where:

- $F_m(x; \theta)$ represents the prediction from individual tree m
- w_m are the weights assigned to each tree's prediction
- θ represents the parameters (structure and splits) of the trees

Bagging approach: Fit trees independently on bootstrap samples, then set $w_m = 1/M$ (equal weights). This assumes that independent fitting will naturally produce diverse trees—but cannot guarantee it.

Boosting approach: Fit trees sequentially, where each tree explicitly corrects the errors of previous trees. The weights w_m emerge from the optimisation process, with more accurate trees receiving higher weights.

The key insight is that when we fit each $F_m(\cdot)$ independently (as in bagging), we cannot ensure they learn different things. Despite being trained on different samples or with different features, trees may end up making similar mistakes because they are drawn to the same dominant patterns in the data.

104 The Boosting Idea

Boosting is an ensemble technique that focuses on minimising a loss function by sequentially adding models that predict the residuals or errors of the ensemble thus far.

Boosting Intuition

1. Fit a weak model to the data
2. Identify where the model makes mistakes (calculate residuals/errors)
3. Fit a new model focusing on those mistakes
4. Add the new model to the ensemble
5. Repeat until some stopping criterion is met

Each iteration corrects mistakes from previous iterations. Stack many **weak learners** (models barely better than random guessing) to build a **strong learner**.

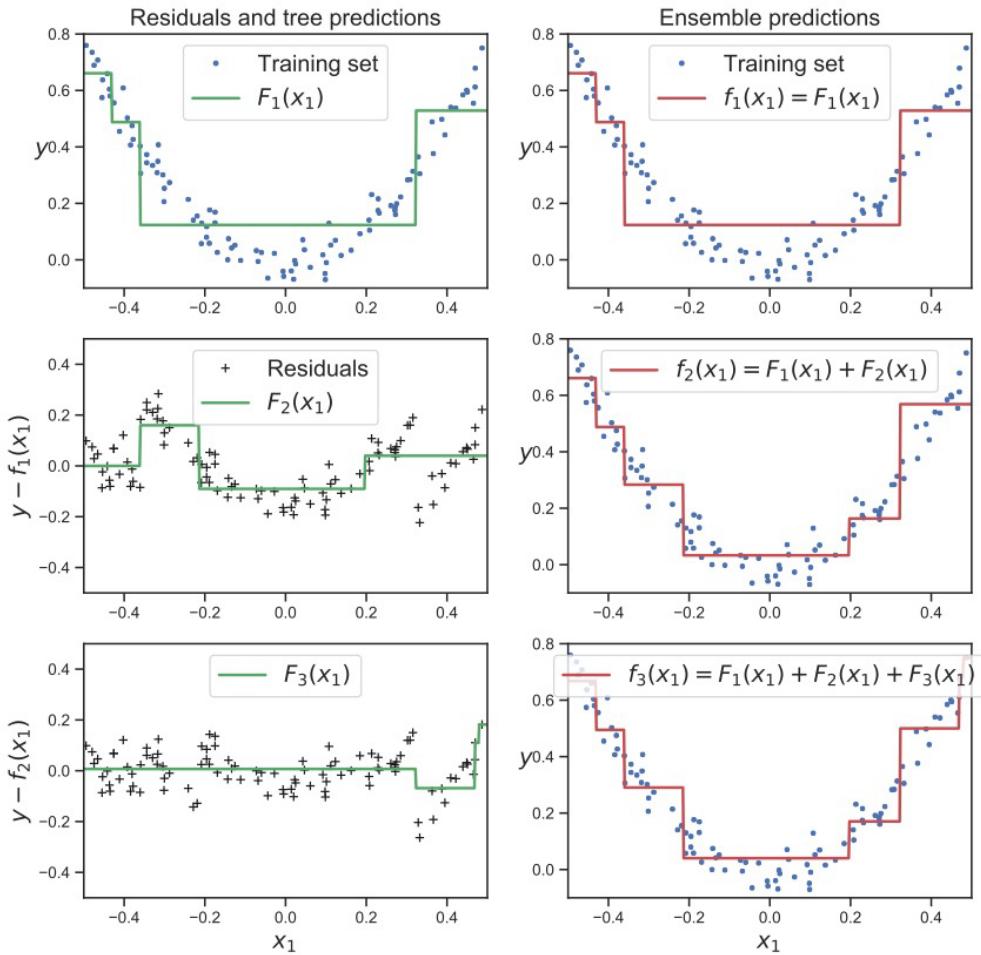


Figure 62: Green: prediction from a single shallow decision tree (weak learner). Red: combined prediction from all trees. Each individual tree is not particularly expressive—often just depth 1–2—but when combined, they capture complex patterns that no single shallow tree could represent. Cross-validation is used to select tree depth and number of iterations.

104.1 Weak Learners and Strong Learners

A central concept in boosting is the distinction between **weak learners** and **strong learners**.

Weak Learner Definition

For binary classification with $y \in \{-1, +1\}$, a **weak learner** is a classifier $h : \mathcal{X} \rightarrow \{-1, +1\}$ satisfying:

$$\Pr[h(x) \neq y] \leq \frac{1}{2} - \gamma$$

for some $\gamma > 0$. The classifier must be better than random guessing by at least γ (the “edge”).

What this means: Random guessing achieves 50% accuracy. A weak learner only needs to do slightly better—say, 51% or 52%. The quantity γ is called the “edge” or “advantage” over random guessing.

Typical weak learners:

- **Decision stumps:** Trees with a single split (depth 1)—the simplest non-trivial tree
- **Shallow trees:** Trees with depth 2–4, capturing limited interactions
- Any simple model: short rules, linear classifiers on subsets of features

The remarkable theoretical result of boosting is that combining weak learners can produce arbitrarily accurate classifiers:

Weak-to-Strong Amplification

If a weak learning algorithm can consistently find classifiers with edge $\gamma > 0$ on any weighted distribution over the training data, then boosting can combine these weak classifiers to achieve arbitrarily low training error.

Intuition: Each weak learner contributes a small improvement. By adaptively reweighting to focus on mistakes, we ensure every weak learner contributes something new. The cumulative effect is a strong classifier.

This is the fundamental theoretical justification for boosting: we do not need powerful individual models. As long as each model is slightly better than random, the combination can be arbitrarily good.

Why use weak learners instead of strong ones? Several compelling reasons:

1. **Speed:** Weak learners are fast to train (stumps require evaluating only p possible splits)
2. **Bias-variance trade-off:** They have high bias but low variance—boosting reduces bias while keeping variance controlled
3. **Robustness:** Strong learners might overfit individual iterations, amplifying noise rather than correcting genuine errors
4. **Complementarity:** Simple models are more likely to capture different aspects of the data, leading to genuine ensemble diversity

The Logic of Boosting

You can often do a better job by iteratively stacking weak learners than by fitting a single complex model. This works because:

- Weak learners are simple and fast to fit
- Each learner specialises in correcting specific errors
- The sequential process naturally decorrelates contributions
- Regularisation (via learning rate) prevents overfitting

104.2 Generic Boosting Loss Function

At iteration m , boosting solves the following optimisation problem:

Generic Boosting Loss at Iteration m

$$(\beta_m, F_m) = \arg \min_{\beta, F} \sum_{i=1}^N \mathcal{L} \left(y_i, \underbrace{f_{m-1}(x_i)}_{\text{previous ensemble}} + \underbrace{\beta}_{\text{learning rate}} \cdot \underbrace{F(x_i)}_{\text{new tree}} \right)$$

Breaking down the terms:

- $f_{m-1}(x_i) = \sum_{j=1}^{m-1} \beta_j F_j(x_i)$ is the prediction from the ensemble at iteration $m-1$ —the sum of all previous trees' contributions
- $F(x_i)$ is the new tree being fitted (with its own internal parameters determining structure and splits)
- β is the learning rate (shrinkage parameter) that weights the new tree's contribution
- $\mathcal{L}(\cdot, \cdot)$ is the loss function (e.g., squared error, exponential loss, log loss)

The learning rate β is crucial: it prevents overfitting by making small adjustments at each iteration rather than large corrections. Think of it as controlling how much we “trust” each new tree. Cross-validation is typically used to select:

- Tree depth (how expressive each weak learner is)
- Number of iterations M (how many weak learners to combine)
- Learning rate β (how much each learner contributes)

104.3 The Double Optimisation Process

The optimisation at each iteration involves two conceptual steps:

Step 1: Fitting the new model to residuals. For each observation i , calculate the residual (or more generally, the negative gradient) from the previous iteration's prediction. Fit a new tree $F(x)$ to these targets. This step focuses on learning from the mistakes of the ensemble thus far.

Step 2: Finding the optimal β . Once $F(x)$ is fitted to predict the targets, find the optimal scaling factor β that minimises the overall loss when this new tree is added to the previous ensemble. This typically involves a line search to find the value of β that best reduces the loss.

In practice, these steps are often simplified: β may be treated as a fixed hyperparameter (set before training begins), and the tree fitting focuses purely on the residuals or pseudo-residuals. Note that the “previous model” f_{m-1} is *all* of the previous trees combined—the ensemble prediction accumulates recursively.

105 Least Squares Boosting

Least squares boosting is a specific form of gradient boosting that uses the squared error loss function—particularly suited to regression problems. It provides the clearest illustration of the “fit to residuals” intuition.

Least Squares Loss

The loss function for least squares boosting is:

$$\ell(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

Inserting this into the generic boosting loss at iteration m :

$$\begin{aligned} \mathcal{L}(y_i, f_{m-1}(x_i) + \beta F(x_i)) &= \frac{1}{2}(y_i - f_{m-1}(x_i) - \beta F(x_i))^2 \\ &= \frac{1}{2} \left(\underbrace{y_i - f_{m-1}(x_i)}_{\text{residual } r_i^{(m)}} - \beta F(x_i) \right)^2 \end{aligned}$$

The new tree F is fitted to predict the **residuals** $r_i^{(m)} = y_i - f_{m-1}(x_i)$ from the previous ensemble.

The interpretation is elegant: at each step, we are essentially predicting the error of the previous model, thereby correcting it. The new tree learns “what the previous ensemble got wrong” and adds a correction.

Least Squares Boosting Procedure

1. Initialise: $f_0(x) = \bar{y}$ (predict the mean—the constant that minimises squared error)
2. For $m = 1, \dots, M$:
 - (a) Compute residuals: $r_i^{(m)} = y_i - f_{m-1}(x_i)$
 - (b) Fit tree F_m to residuals: $F_m = \arg \min_F \sum_i (r_i^{(m)} - F(x_i))^2$
 - (c) Update ensemble: $f_m = f_{m-1} + \beta F_m$

Each tree predicts residuals; each update corrects errors from all previous trees.

Least Squares Boosting = Gradient Boosting with MSE

For squared error loss, the pseudo-residuals *are* the residuals. This is because:

$$-\frac{\partial}{\partial f} \left[\frac{1}{2}(y - f)^2 \right] = y - f = \text{residual}$$

Gradient boosting (which we will see shortly) reduces to sequentially fitting trees to residuals when using squared error loss. This is the most intuitive form of boosting.

106 AdaBoost

AdaBoost (Adaptive Boosting), introduced by Freund and Schapire (1997), was the first practical boosting algorithm. It achieves weak-to-strong amplification through an elegant sample reweighting scheme.

AdaBoost in One Sentence

If the model misclassifies something, weight it up next time!

106.1 Binary Classification Encoding

AdaBoost works with labels y encoded as $\{-1, +1\}$ rather than the more familiar $\{0, 1\}$. This encoding facilitates the mathematics, especially in the context of loss functions and margins.

Label Encoding Transformation

To transform binary labels from $\{0, 1\}$ to $\{-1, +1\}$:

$$\tilde{y} = 2y - 1$$

$$\begin{aligned} y = 0 &\Rightarrow \tilde{y} = 2(0) - 1 = -1 \\ y = 1 &\Rightarrow \tilde{y} = 2(1) - 1 = +1 \end{aligned}$$

Why this encoding? With $y \in \{-1, +1\}$, the product $y \cdot F(x)$ has a natural interpretation:

- $y \cdot F(x) > 0$: correct classification (same sign)
- $y \cdot F(x) < 0$: incorrect classification (opposite sign)
- $|y \cdot F(x)|$: confidence of prediction (larger = more confident)

This product is called the **margin** and appears throughout boosting theory.

AdaBoost models produce predictions $F(x) \in (-\infty, +\infty)$. To interpret these as probabilities, apply a sigmoid function:

$$P(y = 1 | x) = \frac{1}{1 + \exp(-2F(x))}$$

106.2 The Exponential Loss Function

AdaBoost uses the exponential loss function, which heavily penalises misclassifications:

AdaBoost: Exponential Loss

$$\ell(y, F(x)) = \exp(-y \cdot F(x))$$

where $y \in \{-1, +1\}$ and $F(x)$ is the ensemble prediction (a real number).

Understanding the loss:

- If $y = +1$ and $F(x) > 0$: loss = $\exp(-\text{positive}) < 1$ (small loss—correct prediction)
- If $y = +1$ and $F(x) < 0$: loss = $\exp(+\text{positive}) > 1$ (large loss—incorrect prediction)
- If $y = -1$ and $F(x) < 0$: loss = $\exp(-\text{positive}) < 1$ (small loss—correct prediction)
- If $y = -1$ and $F(x) > 0$: loss = $\exp(+\text{positive}) > 1$ (large loss—incorrect prediction)

The loss explodes exponentially when the model is confident and wrong. Correct classifications ($yF(x) > 0$) have loss < 1 , decreasing in margin. Incorrect classifications ($yF(x) < 0$) have loss > 1 , increasing exponentially.

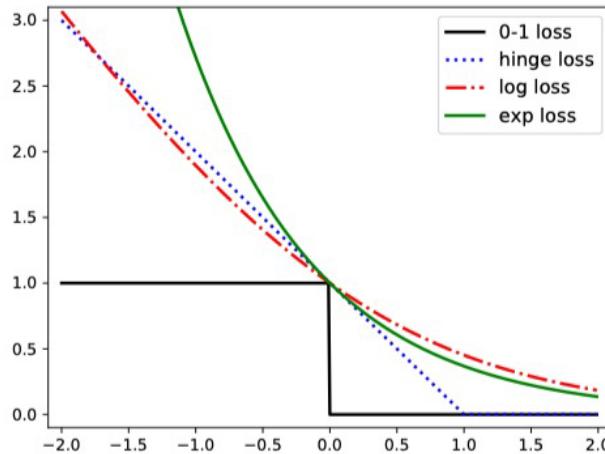


Figure 63: Comparison of loss functions for binary classification. Exponential loss (AdaBoost) penalises errors more aggressively than log loss (logistic regression). The 0-1 loss is discontinuous and non-convex, making it unsuitable for gradient-based optimisation. Both exponential and log loss are differentiable surrogates for 0-1 loss.

106.3 Comparing Loss Functions

Log Loss vs Exponential Loss

Log Loss (Logistic/Cross-Entropy Loss):

$$\ell_{\log}(y, F(x)) = \log(1 + \exp(-2yF(x)))$$

Used in logistic regression. Penalises incorrect predictions, with penalty increasing as discrepancy grows, but **asymptotically linear**—grows at most linearly for very wrong predictions.

Exponential Loss:

$$\ell_{\exp}(y, F(x)) = \exp(-yF(x))$$

Used by AdaBoost. Penalty grows **without bound**—exponentially as predictions move away from actual labels.

Key differences:

- Exponential loss imposes a much higher penalty on large misclassifications
- The exponential penalty drives the model to prioritise correcting misclassifications
- Log loss is more robust to outliers (bounded gradient for large errors)
- Both are effective surrogates for 0-1 loss; from an optimisation perspective, log loss is generally easier to minimise due to its smoother gradient

106.4 The AdaBoost Algorithm

AdaBoost for Binary Classification

Input: Training data $\{(x_i, y_i)\}_{i=1}^N$ with $y_i \in \{-1, +1\}$, number of iterations M

Initialise: Sample weights $w_i^{(1)} = 1/N$ for all i (uniform weights)

For $m = 1, \dots, M$:

1. **Fit weak learner** F_m to training data using weights $w^{(m)}$:

$$F_m = \arg \min_F \sum_{i=1}^N w_i^{(m)} \cdot \mathbf{1}[F(x_i) \neq y_i]$$

(minimise weighted misclassification error)

2. **Compute weighted error rate**:

$$\text{err}_m = \frac{\sum_{i=1}^N w_i^{(m)} \cdot \mathbf{1}[F_m(x_i) \neq y_i]}{\sum_{i=1}^N w_i^{(m)}}$$

3. **Compute learner weight** (how much this tree contributes to ensemble):

$$\beta_m = \frac{1}{2} \log \left(\frac{1 - \text{err}_m}{\text{err}_m} \right)$$

4. **Update sample weights**:

$$w_i^{(m+1)} = w_i^{(m)} \cdot \exp(-\beta_m y_i F_m(x_i))$$

Output: Final classifier

$$f(x) = \text{sign} \left(\sum_{m=1}^M \beta_m F_m(x) \right)$$

Understanding AdaBoost's Moving Parts

- $\beta_m > 0$ when $\text{err}_m < 0.5$ (weak learner is better than random)
- β_m is large when err_m is small (accurate learners get more weight in ensemble)
- Misclassified points ($y_i F_m(x_i) = -1$) have weights multiplied by $e^{\beta_m} > 1$
- Correctly classified points ($y_i F_m(x_i) = +1$) have weights multiplied by $e^{-\beta_m} < 1$
- The algorithm “adapts” by focusing subsequent learners on hard examples

The weight $w_i^{(m)}$ represents how difficult point i has been for the ensemble so far—difficult points accumulate high weights.

106.5 Derivation: AdaBoost as Exponential Loss Minimisation

The AdaBoost algorithm can be derived as **forward stagewise additive modelling** with **exponential loss**. This derivation, due to Friedman, Hastie, and Tibshirani (2000), reveals why the β_m formula takes its particular form.

Derivation of AdaBoost from Exponential Loss

At iteration m , we have ensemble $f_{m-1}(x) = \sum_{j=1}^{m-1} \beta_j F_j(x)$ and seek (β_m, F_m) minimising:

$$L = \sum_{i=1}^N \exp(-y_i [f_{m-1}(x_i) + \beta F(x_i)])$$

Step 1: Factor out the fixed term:

$$L = \sum_{i=1}^N \underbrace{\exp(-y_i f_{m-1}(x_i))}_{=: w_i^{(m)}} \cdot \exp(-y_i \beta F(x_i))$$

The weights $w_i^{(m)} = \exp(-y_i f_{m-1}(x_i))$ depend only on previous iterations—they are fixed when optimising over (β, F) .

Step 2: Since $y_i, F(x_i) \in \{-1, +1\}$, we have $y_i F(x_i) = +1$ if correct, -1 if incorrect:

$$L = \sum_{i:y_i=F(x_i)} w_i^{(m)} e^{-\beta} + \sum_{i:y_i \neq F(x_i)} w_i^{(m)} e^{\beta}$$

Correct predictions contribute $e^{-\beta}$; incorrect predictions contribute e^{β} .

Step 3: Rearrange using $W = \sum_i w_i^{(m)}$ (sum of weights):

$$L = e^{-\beta}(W - W \cdot \text{err}_m) + e^{\beta}W \cdot \text{err}_m = W \left[e^{-\beta}(1 - \text{err}_m) + e^{\beta}\text{err}_m \right]$$

where $\text{err}_m = \frac{\sum_{i:y_i \neq F(x_i)} w_i^{(m)}}{W}$ is the weighted error rate.

Step 4: For fixed F_m , minimise over β by setting $\frac{\partial L}{\partial \beta} = 0$:

$$-e^{-\beta}(1 - \text{err}_m) + e^{\beta}\text{err}_m = 0$$

$$\begin{aligned} e^{2\beta} &= \frac{1 - \text{err}_m}{\text{err}_m} \\ \beta_m &= \frac{1}{2} \log \left(\frac{1 - \text{err}_m}{\text{err}_m} \right) \end{aligned}$$

This is exactly the AdaBoost formula for β_m .

Step 5: For fixed $\beta > 0$, L is minimised when err_m is minimised—i.e., F_m should minimise weighted misclassification error. This is exactly what AdaBoost prescribes.

AdaBoost = Forward Stagewise Exponential Loss

The AdaBoost algorithm is equivalent to greedily minimising exponential loss:

$$\min_f \sum_{i=1}^N \exp(-y_i f(x_i))$$

using forward stagewise additive modelling with weak learners. The weight update $w_i^{(m+1)} = w_i^{(m)} \exp(-\beta_m y_i F_m(x_i))$ maintains $w_i^{(m)} \propto \exp(-y_i f_{m-1}(x_i))$ —the exponential loss of the current ensemble on point i .

The weights naturally encode “how much loss we’ve accumulated on this point.”

Why Exponential Loss Works Well for AdaBoost

- **Weight update mechanism:** The exponential loss directly determines how weights are updated—misclassified observations receive exponentially higher weights
- **Focus on hard cases:** As boosting progresses, the algorithm concentrates on observations that the current ensemble finds most challenging
- **Adaptive learning:** The name “Adaptive Boosting” reflects this dynamic focus on difficult examples
- **Closed-form β_m :** The exponential loss allows us to derive β_m analytically, avoiding numerical optimisation

Where we had large loss before, we weight that up by the size of that loss. The exponential loss naturally implements this reweighting.

106.6 Convergence Guarantees

AdaBoost has remarkable theoretical guarantees on training error:

Training Error Bound

After M iterations of AdaBoost, the training error satisfies:

$$\frac{1}{N} \sum_{i=1}^N \mathbf{1}[\text{sign}(f_M(x_i)) \neq y_i] \leq \exp \left(-2 \sum_{m=1}^M \gamma_m^2 \right)$$

where $\gamma_m = \frac{1}{2} - \text{err}_m$ is the “edge” of learner m over random guessing.

What this means: The bound involves the sum of squared edges. Each weak learner contributes γ_m^2 to the exponent.

Corollary: If each weak learner achieves edge $\gamma_m \geq \gamma > 0$, then:

$$\text{Training error} \leq \exp(-2M\gamma^2)$$

Training error decreases **exponentially** in M . With enough iterations, training error $\rightarrow 0$.

This is a remarkable guarantee: as long as each weak learner is slightly better than random (edge $\gamma > 0$), we can drive training error to zero exponentially fast. This is the formal statement of “weak-to-strong amplification.”

NB!

Zero training error does not mean zero test error! AdaBoost can overfit, especially with many iterations and expressive weak learners. The exponential loss also makes AdaBoost **sensitive to outliers and label noise**—mislabelled points get exponentially increasing weight, potentially dominating the learning process.

A single noisy observation that is consistently misclassified can accumulate enormous weight, distorting the entire ensemble.

106.7 Worked Example: AdaBoost Step-by-Step

To solidify understanding, let us trace through AdaBoost on a simple example.

AdaBoost Example Dataset

Consider 10 points with 1 feature:

i	1	2	3	4	5	6	7	8	9	10
x_i	1	2	3	4	5	6	7	8	9	10
y_i	+1	+1	+1	-1	-1	-1	+1	+1	+1	-1

No single threshold perfectly separates the classes (the +1s are split into two groups: $\{1, 2, 3\}$ and $\{7, 8, 9\}$).

Iteration 1:

Initial weights: $w_i^{(1)} = 0.1$ for all i (uniform).

We search for the best decision stump. Consider $F_1(x) = +1$ if $x \leq 3.5$, else -1 .

Predictions: $\hat{y} = (+1, +1, +1, -1, -1, -1, -1, -1, -1)$

Comparing with true labels: $y = (+1, +1, +1, -1, -1, -1, +1, +1, +1, -1)$

Misclassified: points 7, 8, 9 (predicted -1 , true $+1$).

$$\text{err}_1 = 0.1 + 0.1 + 0.1 = 0.3$$

$$\beta_1 = \frac{1}{2} \log \left(\frac{1 - 0.3}{0.3} \right) = \frac{1}{2} \log \left(\frac{0.7}{0.3} \right) \approx 0.424$$

Update weights:

- Correct predictions (points 1–6, 10): $w_i^{(2)} = 0.1 \cdot e^{-0.424} \approx 0.0655$
- Incorrect predictions (points 7, 8, 9): $w_i^{(2)} = 0.1 \cdot e^{0.424} \approx 0.1528$

After normalising (so weights sum to 1):

i	1	2	3	4	5	6	7	8	9	10
$w_i^{(2)}$	0.071	0.071	0.071	0.071	0.071	0.071	0.167	0.167	0.167	0.071

Points 7, 8, 9 now have higher weight—the next weak learner will focus on getting these right.

Iteration 2:

With the new weights, the best stump changes. Consider $F_2(x) = +1$ if $x \geq 6.5$, else -1 .

Predictions: $\hat{y} = (-1, -1, -1, -1, -1, -1, +1, +1, +1, +1)$

Comparing with true labels: Misclassified are points 1, 2, 3 (predicted -1 , true $+1$) and point 10 (predicted $+1$, true -1).

Weighted error: $\text{err}_2 = 3 \times 0.071 + 1 \times 0.071 = 0.284$

$$\beta_2 = \frac{1}{2} \log \left(\frac{0.716}{0.284} \right) \approx 0.462$$

The ensemble after 2 iterations:

$$f_2(x) = 0.424 \cdot F_1(x) + 0.462 \cdot F_2(x)$$

Let us check some predictions:

- For $x = 7$: $f_2(7) = 0.424 \cdot (-1) + 0.462 \cdot (+1) = 0.038 > 0 \Rightarrow$ predict $+1$ (correct!)
- For $x = 4$: $f_2(4) = 0.424 \cdot (-1) + 0.462 \cdot (-1) = -0.886 < 0 \Rightarrow$ predict -1 (correct!)
- For $x = 10$: $f_2(10) = 0.424 \cdot (-1) + 0.462 \cdot (+1) = 0.038 > 0 \Rightarrow$ predict $+1$ (incorrect—but just barely!)

Worked Example Insights

- Neither stump alone correctly classifies all points
- The weighted combination achieves better accuracy than either stump alone
- Weights adaptively focus on misclassified points
- More iterations would continue improving (eventually zero training error)
- The stumps “vote” with different weights, and their combination captures patterns neither could alone

107 Gradient Boosting

AdaBoost is elegant but limited to exponential loss. **Gradient boosting**, introduced by Friedman (2001), generalises boosting to work with *any* differentiable loss function.

Gradient Boosting: The Core Insight

Gradient boosting performs **gradient descent in function space**:

- Ordinary gradient descent: update parameters $\theta \leftarrow \theta - \eta \nabla_{\theta} \mathcal{L}$
- Gradient boosting: update function $f \leftarrow f + \eta \cdot h$, where h approximates $-\nabla_f \mathcal{L}$

We cannot directly compute gradients over the infinite-dimensional space of all functions, so we approximate the gradient direction using a tree fitted to the **negative gradient** at each training point.

107.1 Relationship to Other Methods

It is helpful to understand how the boosting methods relate to each other:

- **Least Squares Boosting:** A specific instance of gradient boosting using squared error loss. Particularly suited to regression problems. The negative gradient equals the residual.
- **AdaBoost:** Another specific instance, using exponential loss. Designed for classification with focus on reweighting misclassified instances. The negative gradient leads to the sample reweighting scheme.
- **Gradient Boosting:** The general framework. Its use of gradient descent on the loss function provides a systematic and generalisable approach to minimising prediction error with any differentiable loss.

107.2 Gradient Descent in Function Space

Gradient Boosting Framework

Objective: Find a function f^* that minimises the loss $\mathcal{L}(f) = \sum_i \ell(y_i, f(x_i))$.

Gradient of Loss: The gradient of the loss with respect to the predictions points in the direction of steepest increase. By moving in the opposite direction, we reduce the loss:

$$g_i^{(m)} = \frac{\partial \ell(y_i, f(x_i))}{\partial f(x_i)} \Big|_{f=f_{m-1}}$$

Pseudo-residuals (negative gradient): The target for the new tree is:

$$r_i^{(m)} = -g_i^{(m)} = -\frac{\partial \ell(y_i, f(x_i))}{\partial f(x_i)} \Big|_{f=f_{m-1}}$$

These “pseudo-residuals” point in the direction we need to move our predictions to reduce loss.

Update rule:

$$f_m = f_{m-1} + \eta \cdot F_m$$

where F_m is fitted to approximate the pseudo-residuals $\{r_i^{(m)}\}$.

Gradient Boosting as Gradient Descent

There is a beautiful analogy between ordinary gradient descent and gradient boosting:

	Standard Gradient Descent	Gradient Boosting
Space	Parameter space \mathbb{R}^P	Function space $\{f : \mathcal{X} \rightarrow \mathbb{R}\}$
Object	Parameter vector θ	Function f
Update	$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$	$f \leftarrow f + \eta F$
Gradient	Computed analytically	Approximated by tree F
Step size	Learning rate η	Shrinkage factor η

In gradient boosting, we cannot directly add the gradient to our function (we do not have an explicit parametric representation). Instead, we fit a tree F to approximate the negative gradient, then add that tree to our ensemble.

107.3 The Generic Algorithm

Gradient Boosting Algorithm

Input: Training data $\{(x_i, y_i)\}_{i=1}^N$, differentiable loss $\mathcal{L}(y, f)$, number of iterations M , learning rate η

Initialise: $f_0(x) = \arg \min_c \sum_{i=1}^N \mathcal{L}(y_i, c)$ (constant prediction minimising loss)

For $m = 1, \dots, M$:

1. **Compute pseudo-residuals** (negative gradient at each point):

$$r_i^{(m)} = - \left[\frac{\partial \mathcal{L}(y_i, f(x_i))}{\partial f(x_i)} \right]_{f=f_{m-1}}$$

2. **Fit weak learner** F_m to pseudo-residuals:

$$F_m = \arg \min_F \sum_{i=1}^N \left(r_i^{(m)} - F(x_i) \right)^2$$

3. **Line search** (optional): find optimal step size

$$\rho_m = \arg \min_\rho \sum_{i=1}^N \mathcal{L}(y_i, f_{m-1}(x_i) + \rho \cdot F_m(x_i))$$

4. **Update ensemble:**

$$f_m(x) = f_{m-1}(x) + \eta \cdot \rho_m \cdot F_m(x)$$

Output: Final model $f_M(x)$

The key insight is Step 1: the **pseudo-residuals** $r_i^{(m)}$ point in the direction of steepest descent for each training point. By fitting a tree to these pseudo-residuals, we approximate the gradient direction in function space with a function we can actually represent.

107.4 Gradient Boosting for Regression (Squared Error)

For regression with squared error loss $\mathcal{L}(y, f) = \frac{1}{2}(y - f)^2$:

Gradient Boosting with MSE Loss

The negative gradient is:

$$r_i^{(m)} = -\frac{\partial}{\partial f} \left[\frac{1}{2} (y_i - f)^2 \right]_{f=f_{m-1}(x_i)} = y_i - f_{m-1}(x_i)$$

These are exactly the **residuals**—the difference between true values and current predictions.

Algorithm simplifies to:

1. Compute residuals: $r_i^{(m)} = y_i - f_{m-1}(x_i)$
2. Fit tree F_m to residuals
3. Update: $f_m = f_{m-1} + \eta \cdot F_m$

This is the “fit to residuals” intuition that originally motivated boosting for regression.

107.5 Gradient Boosting for Classification (Log Loss)

For binary classification with log loss (logistic regression loss):

$$\mathcal{L}(y, f) = \log(1 + \exp(-y \cdot f))$$

where $y \in \{-1, +1\}$ and $f(x)$ is the log-odds.

Gradient Boosting with Log Loss

The negative gradient is:

$$r_i^{(m)} = -\frac{\partial \mathcal{L}(y_i, f)}{\partial f} \Big|_{f=f_{m-1}(x_i)} = \frac{y_i}{1 + \exp(y_i f_{m-1}(x_i))}$$

Interpretation:

- Points with high loss (wrong predictions with high confidence) have pseudo-residuals close to ± 1
- Points already correctly classified with high confidence have pseudo-residuals close to 0

The tree F_m is fitted to these pseudo-residuals, focusing on hard-to-classify points.

Final predictions are converted to probabilities:

$$\Pr(Y = +1 | x) = \frac{1}{1 + \exp(-f_M(x))}$$

107.6 Connection to AdaBoost

AdaBoost as Gradient Boosting with Exponential Loss

For exponential loss $\mathcal{L}(y, f) = \exp(-yf)$:

Negative gradient:

$$r_i^{(m)} = -\frac{\partial}{\partial f} \exp(-y_i f) \Big|_{f=f_{m-1}(x_i)} = y_i \exp(-y_i f_{m-1}(x_i))$$

Up to a constant factor, $|r_i^{(m)}| \propto \exp(-y_i f_{m-1}(x_i)) = w_i^{(m)}$ (the AdaBoost weights).

Conclusion: AdaBoost is gradient boosting with exponential loss, where the weak learner is constrained to output ± 1 and the step size is computed analytically. The sample reweighting in AdaBoost emerges naturally from the gradient of exponential loss.

NB!

Exponential loss is sensitive to outliers because it increases unboundedly for large negative margins. Log loss (used in gradient boosting for classification) is more robust—it increases linearly rather than exponentially for misclassified points. For noisy data with potential label errors, gradient boosting with log loss often outperforms AdaBoost.

108 XGBoost and LightGBM

XGBoost (Extreme Gradient Boosting) and LightGBM are highly optimised implementations of gradient boosting that dominate machine learning competitions on tabular data.

Why XGBoost/LightGBM Are So Successful

1. **Regularisation:** Explicit penalty on tree complexity prevents overfitting
2. **Second-order optimisation:** Uses both gradient and Hessian for better convergence
3. **Efficient implementation:** Histogram-based splits, parallel tree construction, out-of-core computation
4. **Handling missing values:** Learns optimal direction for missing values at each split
5. **Built-in cross-validation:** Early stopping based on validation performance

The name “extreme” refers to pushing the limits of gradient boosting performance.

108.1 Regularised Objective

XGBoost Objective Function

XGBoost minimises a regularised objective at each iteration:

$$\mathcal{J}^{(m)} = \sum_{i=1}^N \mathcal{L}(y_i, f_{m-1}(x_i) + h_m(x_i)) + \Omega(h_m)$$

where the regularisation term penalises tree complexity:

$$\Omega(h) = \gamma J + \frac{1}{2} \lambda \sum_{j=1}^J w_j^2$$

Breaking down the regularisation:

- J = number of leaves in tree h
- w_j = prediction (weight) at leaf j
- γ = penalty for adding a leaf (encourages simpler trees with fewer leaves)
- λ = L2 regularisation on leaf weights (shrinks predictions towards zero)

This provides more continuous control over tree complexity than simply limiting depth. The γ parameter essentially asks: “Is this split worth adding another leaf?”

108.2 Second-Order Approximation

Standard gradient boosting uses only the gradient (first derivative). XGBoost uses a second-order Taylor expansion, incorporating the Hessian (second derivative):

Second-Order Taylor Expansion

Expand the loss around f_{m-1} :

$$\mathcal{L}(y_i, f_{m-1}(x_i) + h(x_i)) \approx \mathcal{L}(y_i, f_{m-1}(x_i)) + g_i h(x_i) + \frac{1}{2} h_i h(x_i)^2$$

where:

- $g_i = \frac{\partial \mathcal{L}(y_i, f)}{\partial f} \Big|_{f_{m-1}(x_i)}$ (gradient—first derivative)
- $h_i = \frac{\partial^2 \mathcal{L}(y_i, f)}{\partial f^2} \Big|_{f_{m-1}(x_i)}$ (Hessian—second derivative)

For a tree with leaves R_1, \dots, R_J and leaf weights w_1, \dots, w_J , the objective becomes:

$$\mathcal{J}^{(m)} \approx \text{const} + \sum_{j=1}^J \left[G_j w_j + \frac{1}{2} (H_j + \lambda) w_j^2 \right] + \gamma J$$

where $G_j = \sum_{i \in R_j} g_i$ and $H_j = \sum_{i \in R_j} h_i$ are the sum of gradients and Hessians for points in leaf j .

Optimal leaf weight (minimising the quadratic in w_j):

$$w_j^* = -\frac{G_j}{H_j + \lambda}$$

Optimal objective value (for fixed tree structure):

$$\mathcal{J}^* = -\frac{1}{2} \sum_{j=1}^J \frac{G_j^2}{H_j + \lambda} + \gamma J$$

The second-order approximation corresponds to Newton's method rather than gradient descent—it uses curvature information to take better steps, often converging faster than first-order methods.

Split Gain in XGBoost

When evaluating a split, XGBoost computes the gain:

$$\text{Gain} = \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] - \gamma$$

What this means:

- First term: objective with left child alone
- Second term: objective with right child alone
- Third term: objective with parent (no split)
- γ : penalty for adding complexity

A split is only made if $\text{Gain} > 0$ (must overcome the γ penalty for adding complexity). This provides automatic pruning during tree construction.

XGBoost vs Random Forests

	Random Forests	XGBoost
Base strategy	Bagging	Boosting
Tree fitting	Independent, parallel	Sequential, corrective
Tree depth	Typically deep (unlimited)	Typically shallow (3–6)
Variance reduction	Averaging independent trees	Regularisation
Bias reduction	Limited	Strong (sequential correction)
Feature sampling	At each node	At each node (optional)
Interpretability	Moderate	Lower (more trees, interactions)

Random Forests is to bagging what XGBoost is to boosting. Both use feature subsampling at nodes, but their fundamental strategies differ: Random Forests builds diverse trees in parallel, while XGBoost builds complementary trees sequentially.

108.3 LightGBM Innovations

LightGBM (Light Gradient Boosting Machine) introduces additional optimisations that make it faster than XGBoost on large datasets:

LightGBM Key Features

- **Leaf-wise growth:** Splits the leaf with highest gain globally (vs XGBoost's level-wise growth, which splits all leaves at each depth before proceeding). Leads to faster convergence but higher overfitting risk—may need more regularisation.
- **Histogram-based splits:** Buckets continuous features into discrete bins (e.g., 256 bins). Reduces split evaluation from $O(n)$ to $O(\text{bins})$. Speeds up training dramatically with minimal accuracy loss.
- **Gradient-based one-side sampling (GOSS):** Keeps all points with large gradients (hard examples); randomly samples points with small gradients (easy examples). Focuses computation on informative points.
- **Exclusive feature bundling (EFB):** Bundles mutually exclusive sparse features (features that are rarely non-zero simultaneously) to reduce effective dimensionality.

109 Hyperparameters and Tuning

Gradient boosting has many hyperparameters. Understanding them is crucial for achieving good performance and avoiding overfitting.

Critical Hyperparameters

Number of iterations (M / `n_estimators`):

- More iterations = lower bias, higher variance (more capacity to fit training data)
- Too many iterations \Rightarrow overfitting
- **Best practice:** Use early stopping—train until validation error stops improving

Learning rate (η / `learning_rate`):

- Shrinks each tree's contribution: $f_m = f_{m-1} + \eta \cdot F_m$
- Smaller η = more iterations needed, but often better generalisation
- Typical values: 0.01–0.3
- **Rule of thumb:** $\eta \times M \approx \text{constant}$ for similar final models

Tree depth (`max_depth`):

- Shallow trees (depth 1–6) work best for boosting
- Depth 1 = stumps (no interactions); depth 2 = pairwise interactions; etc.
- Deeper trees increase variance and overfitting risk
- **Default:** 3–6 for gradient boosting (vs unlimited for random forests)

Subsampling (`subsample`):

- Fraction of training data used per iteration
- Values < 1 add stochasticity (“stochastic gradient boosting”)
- Reduces variance and computation; typical values: 0.5–1.0

Hyperparameter Tuning Strategy

A systematic approach to tuning:

1. **Set learning rate small** (0.01–0.1) and use early stopping to find M
2. **Tune tree depth:** Start with 3–6; shallow is usually better
3. **Tune regularisation:** γ (min split gain), λ (L2 on weights)
4. **Add subsampling:** Column and row subsampling (0.6–1.0)
5. **Increase learning rate** if training is too slow (and reduce M accordingly)

Use cross-validation or a held-out validation set. Grid search, random search, or Bayesian optimisation for systematic tuning.

Stochastic Gradient Boosting

Setting `subsample < 1` and/or `colsample_bytree < 1` introduces randomness similar to random forests:

- `subsample`: Each tree trained on a random subset of rows
- `colsample_bytree`: Each tree considers a random subset of columns
- `colsample_bylevel`: Random columns at each depth level
- `colsample_bynode`: Random columns at each split

This reduces correlation between trees, decreasing variance. Combined with learning rate shrinkage, stochastic gradient boosting is highly effective and combines benefits of both bagging (randomness) and boosting (sequential correction).

Early Stopping

Train while monitoring validation error. Stop when validation error hasn't improved for k consecutive iterations ("patience").

In **XGBoost**/**LightGBM**:

```
model.fit(X_train, y_train,
          eval_set=[(X_val, y_val)],
          early_stopping_rounds=10)
```

This automatically determines the optimal number of iterations, avoiding the need to guess M . The model state at the best iteration is retained.

110 Model Interpretation

Tree-based ensemble methods, while powerful, can be difficult to interpret. Two key techniques help us understand what these models have learned: feature importance (which features matter) and partial dependence (how features affect predictions).

110.1 Feature Importance

Feature Importance

Feature importance quantifies the contribution of each feature to the predictive power of the model. For tree-based models, a common measure is the total gain attributable to each feature:

$$R_k(T) = \sum_{j=1}^{J-1} G_j \cdot \mathbf{1}[v_j = k]$$

where:

- $R_k(T)$ is the importance of feature k in tree T
- J is the number of nodes (internal + leaves) in the tree
- G_j is the gain in accuracy (or reduction in impurity) at node j
- v_j is the feature used for splitting at node j
- $\mathbf{1}[v_j = k]$ is 1 if node j splits on feature k , 0 otherwise

To get overall importance, sum (or average) over all trees and normalise:

$$R_k = \sum_{m=1}^M R_k(T_m)$$

Normalise so that $\sum_k R_k = 100$ (or 1) for interpretability.

Alternative measures:

- **Frequency-based importance:** Count how often each feature is used for splitting
- **Permutation importance:** Measure accuracy drop when feature k is randomly permuted (model-agnostic; applies to any model)

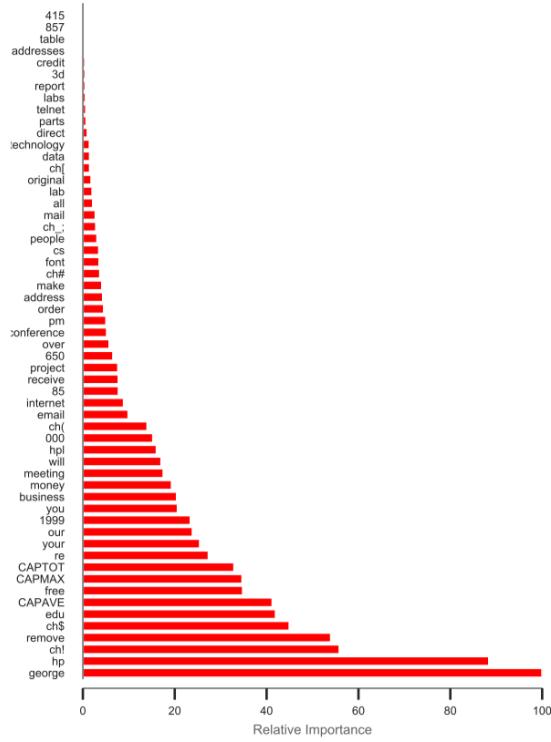


Figure 64: Feature importance for spam classification, where features are word occurrences. “George” is highly predictive—when the model splits on this feature, accuracy improves substantially. But importance alone does not tell us whether “George” indicates spam or not-spam.

NB!

Feature importance tells you **which** features matter, not **how** they affect predictions. A feature can be important but have different effects depending on context and interactions with other features.

In the spam example, “George” is important, but we cannot say whether its presence makes emails more or less likely to be spam. Due to interaction effects, it might increase spam probability in some contexts and decrease it in others. For causal or directional interpretation, use partial dependence plots or SHAP values.

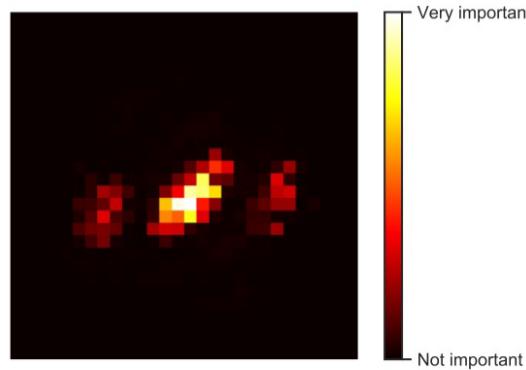


Figure 65: Feature importance for digit classification (3 vs 8), where features are pixel values. Intuitively, the central pixels are most useful for distinguishing these digits—the middle vertical stroke differs between 3 and 8.

110.2 Partial Dependence Plots

To understand *how* a feature affects predictions, we use partial dependence plots.

Partial Dependence Plot

A partial dependence plot shows how the average prediction changes as one feature varies, marginalising over all other features:

$$\bar{f}(x_k) = \frac{1}{N} \sum_{i=1}^N f(x_1^{(i)}, \dots, x_{k-1}^{(i)}, x_k, x_{k+1}^{(i)}, \dots, x_p^{(i)})$$

Procedure:

1. Choose a feature k and a value c
2. For each observation i in the dataset:
 - Create a modified observation where feature k is set to c
 - Keep all other features at their original values
 - Compute the model prediction
3. Average all these predictions to get $\bar{f}(x_k = c)$
4. Repeat for a range of values c and plot

This traces out the marginal effect of feature k on predictions, averaging over the joint distribution of other features.

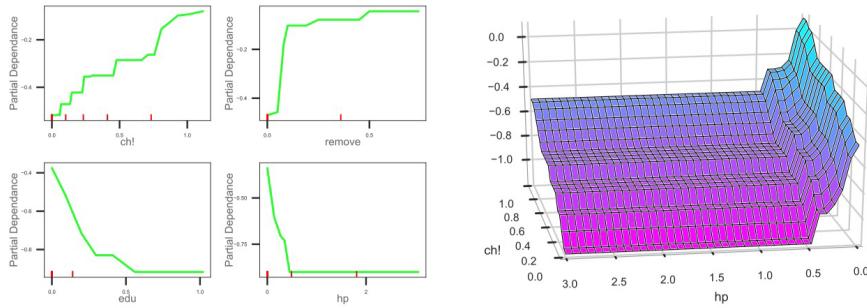


Figure 66: Partial dependence plots showing how predictions change as individual features vary. Each curve shows the average prediction when that feature is set to a specific value, averaging over all other features. The shape reveals the relationship between feature and prediction.

Interpreting Partial Dependence

- Upward slope: increasing the feature increases the prediction
- Downward slope: increasing the feature decreases the prediction
- Flat regions: the feature has little effect in that range
- Non-monotonic patterns: complex, potentially non-linear relationships

Unlike feature importance, partial dependence reveals the *direction* and *shape* of the relationship between features and predictions.

NB!

Partial dependence plots can be misleading when features are correlated. Setting a feature to an unusual value while keeping correlated features at their original values may create unrealistic combinations that never occur in practice.

For example, setting “square footage” to a very high value while keeping “number of bedrooms” at 1 creates an unrealistic house. The model’s prediction for this impossible configuration may not be meaningful.

Consider using Individual Conditional Expectation (ICE) plots (which show one line per observation) or SHAP values for more nuanced interpretation.

110.3 Interpretability vs Performance Trade-off

Interpretability Trade-off

Method	Interpretability	Performance	Use Case
Single tree	High	Low–Medium	Regulatory, explainable AI
Random forest	Medium	High	Robust default
Gradient boosting	Low	Highest	Competitions, production

Boosted ensembles are often black boxes—hundreds of trees combined make it impossible to trace a single decision path.

Interpretability tools for complex ensembles:

- Feature importance (gain-based, permutation)
- Partial dependence plots
- SHAP values (local explanations with theoretical guarantees)
- Surrogate models (fit interpretable model to boosted predictions)

111 Practical Considerations

111.1 When Boosting Overfits

NB!

Boosting can overfit, especially when:

- **Too many iterations:** Training error continues decreasing while test error increases
- **Noisy labels:** Boosting focuses on hard examples—which may be mislabelled noise rather than genuinely difficult cases
- **Deep trees:** More capacity per tree = faster overfitting
- **Large learning rate:** Big steps can oscillate past the optimum
- **Small datasets:** More iterations than data points leads to memorisation

Mitigation strategies:

- Early stopping based on validation error
- Small learning rate with many iterations
- Shallow trees (depth 3–6)
- Subsampling (rows and columns)—adds randomness
- Regularisation (γ, λ in XGBoost)

111.2 Comparison with Neural Networks

Practical Takeaways

Tree-based models are workhorses. Random Forests and XGBoost are robust, scalable, and amenable to tuning (many useful hyperparameters). They should be your baseline before trying more complex models.

Many production models are boosted models. Unlike neural networks, they are much easier to tune: even if you get the learning rate slightly wrong, they still work. A neural network with a suboptimal learning rate may fail to converge entirely.

Rarely should you use something more complex without benchmarking against them first. For structured/tabular data, tree ensembles remain highly competitive with deep learning approaches. Neural networks excel on unstructured data (images, text, audio), but gradient boosting often wins on tabular data.

112 Summary

Key Concepts from Week 8: Boosting

1. **Bagging vs Boosting:** Bagging reduces variance (parallel, independent trees); boosting reduces bias (sequential, corrective trees)
2. **Correlated errors:** Bagging's effectiveness is limited by correlation between tree errors; boosting explicitly decorrelates by targeting mistakes
3. **Weak learners:** Models barely better than random; combined to form strong learners. The edge γ over random guessing is all that matters.
4. **AdaBoost:** Reweighting misclassified samples; equivalent to exponential loss minimisation; training error decreases exponentially in number of iterations
5. **Gradient boosting:** Fit trees to negative gradient of any differentiable loss; gradient descent in function space
6. **For MSE:** Gradient = residual; fit trees to residuals
7. **For classification:** Use log loss for robustness (vs exponential loss in AdaBoost)
8. **XGBoost/LightGBM:** Regularised objective + second-order approximation + computational optimisations
9. **Key hyperparameters:** Learning rate (small), iterations (early stopping), tree depth (shallow), subsampling
10. **Overfitting:** More likely than random forests; mitigate with early stopping, regularisation, shallow trees
11. **Interpretation:** Feature importance (which features), partial dependence (how features affect predictions), SHAP values (local explanations)

Practical Recommendations

- **Start with XGBoost or LightGBM** for tabular data—often achieves best performance
- **Use early stopping**—don't manually set number of iterations
- **Small learning rate** (0.01–0.1) with patience
- **Shallow trees** (depth 3–6)—boosting doesn't need deep trees
- **Add regularisation** if overfitting (γ , λ , subsampling)
- **Compare to random forest**—if similar performance, RF may be preferable (more robust, parallelisable, less tuning)

NB!

When boosting can fail:

- Very noisy data with many outliers (especially AdaBoost with exponential loss)
- When the true relationship is best captured by linear models (boosted trees may overfit)
- When interpretability is paramount (consider simpler models or careful use of explanation tools)
- When data has strong temporal dependencies (consider specialised time series models)
- Very small datasets where the sequential process may memorise noise

Tree-based methods (random forests, XGBoost) are robust, scalable, and easy to tune. They should be your baseline for tabular data before trying more complex models like neural networks.

113 Overview

Chapter Summary

This chapter addresses a fundamental question: **which data should we collect?** When labelling is expensive, strategic sampling can dramatically reduce costs while maintaining model quality. We cover:

- **Data leakage:** The silent killer of ML models—using information unavailable at prediction time
- **Sampling schemes:** Random, stratified, cluster, and systematic sampling
- **Importance sampling:** Reweighting samples from one distribution to estimate expectations under another
- **Active learning:** Iteratively selecting the most informative points to label
- **Leverage score sampling:** Exploiting linear algebra for efficient regression
- **Random Fourier Features:** Efficient kernel approximations
- **Multi-armed bandits:** Balancing exploration and exploitation for reward maximisation
- **AIPW:** Combining models with sampling corrections for unbiased prevalence estimation

How should we choose which data to collect? This question can be approached from three distinct perspectives, each addressing a different goal:

1. **Sampling for better models:** Active learning, leverage score sampling
2. **Sampling for better decisions:** Multi-armed bandits
3. **Sampling to measure prevalence:** AIPW

The Setup

We have N observations of features X , but measuring labels y is expensive.

Examples:

- **Hate speech detection:** X (text) is free to obtain; y (is it hate speech?) requires human judgement
- **Medical diagnosis:** X (symptoms, basic tests) is cheap; y (diagnosis) requires expensive tests or specialist review
- **Drug discovery:** X (molecular structure) is known or computable; y (biological activity) requires costly lab experiments
- **Satellite imagery:** X (image) is available; y (land use classification) requires expert annotation

Core Question: How should we choose which $n \ll N$ observations to label?

Iterative Formulation: Alternatively, suppose we already have n_0 labelled observations $\{(X_i, y_i)\}_{i=1}^{n_0}$. Based on our current model, how should we choose the next n_1 observations to label, using only their features X ?

11.3.1 Explanation vs Prediction

Before diving into sampling strategies, it is worth distinguishing two fundamentally different objectives in machine learning, as they lead to different sampling considerations:

Explanation: The objective is to understand the causal impact of feature(s) A on outcome Y . We seek to uncover causal relationships or explain variations in the outcome. In sampling, this might involve selecting samples that provide the best information about causal relationships, often requiring careful design to avoid confounding factors.

Prediction: The objective is to predict an unseen outcome Y based on observed features X . We aim to build models that generalise well to new, unseen data. The focus is on optimising predictive accuracy rather than understanding mechanisms.

Two Paths to Better Data

For explanation:

- Improve causal inferences from existing data: Observational methods
- Improve the data used for causal inference: Experimentation

For prediction:

- Improve models for prediction: Most of this course
- Improve the data used for prediction: This chapter

114 Data Leakage

Section Summary

Data leakage occurs when training uses information unavailable at prediction time, leading to overly optimistic performance estimates that collapse in production. Common sources include **temporal leakage** (using future to predict past), **target leakage** (features derived from the outcome), and **preprocessing leakage** (fitting transformations before splitting). Prevention requires understanding the deployment context and ensuring train/test splits mirror production conditions.

Data leakage is one of the most insidious problems in applied machine learning. Models with data leakage may appear to perform exceptionally well during training and testing, but fail dramatically on truly unseen data. The problem is subtle because the model genuinely learns something—it just learns patterns that will not be available in production.

Definition: Data Leakage

Data leakage occurs when information is used during model training that would not be legitimately available when the model makes predictions in production.

Formally, if $\mathcal{I}_{\text{train}}$ denotes the information available during training and $\mathcal{I}_{\text{deploy}}$ denotes the information available at deployment:

$$\mathcal{I}_{\text{train}} \supsetneq \mathcal{I}_{\text{deploy}} \implies \text{leakage}$$

The consequence is that empirical performance estimates (cross-validation, test set error) are **optimistically biased**—the model appears better than it actually is.

Unpacking the definition: The set inclusion $\mathcal{I}_{\text{train}} \supsetneq \mathcal{I}_{\text{deploy}}$ means that training has access to strictly more information than deployment. This could be:

- **Direct access:** A feature that exists in training data but will not exist at prediction time
- **Indirect access:** Information encoded in other features, preprocessing statistics, or the structure of the train/test split itself
- **Temporal access:** Knowledge of what happens after the prediction point

NB!

Data leakage is insidious because it often produces models that appear to work brilliantly during development but fail catastrophically in production. Unlike other modelling errors, leakage can be difficult to detect from performance metrics alone—the test set error may look excellent precisely because it suffers from the same leakage as training.

The fundamental question: “Would this information be available at the time I need to make a prediction?”

114.1 Types of Data Leakage

Understanding the different forms leakage can take helps in prevention and detection.

114.1.1 Temporal Leakage

Using information from the future to predict the past. This is the most common form of leakage in time-dependent problems.

Example: Stock Price Prediction

Suppose we want to predict whether a stock will go up tomorrow. A naive approach might include features like:

- Moving averages computed over windows that include future data
- Analyst ratings that were updated after the prediction date
- Market indices from the same day (if predicting opening from closing)

Even a simple random train/test split causes leakage: if training includes data from 2023 and testing includes data from 2022, the model has “seen the future.”

Prevention: Always use a **temporal split**. Training data must strictly precede test data. See Week 4 for time series cross-validation.

Why temporal leakage is so common: Many datasets are created retrospectively, with all features computed at once. It is easy to include a feature that, while technically available in the historical record, would not have been known at the time predictions were needed. A classic example: using end-of-quarter financial reports to predict stock movements during that quarter.

114.1.2 Target Leakage

Features that are derived from or highly correlated with the target variable, often because they are caused by the target rather than causing it.

Example: Medical Diagnosis

Predicting whether a patient has disease Y :

- Feature: “Treatment for Y prescribed” — this is caused by the diagnosis!
- Feature: “Referred to specialist for Y ” — same problem
- Feature: “Blood marker Z ” where Z is only measured when Y is suspected

These features have high predictive power in training but will not be available before diagnosis in production.

Prevention: For each feature, ask: “Would this feature be available at the time the prediction needs to be made?”

The causal structure of target leakage: In target leakage, the problematic feature is typically a *consequence* of the target, not a *cause*. The causal graph looks like:

$$\text{Target } Y \longrightarrow \text{Feature } X$$

The model learns the reverse arrow (predicting Y from X), which works perfectly in training but fails when X is not yet observed because Y has not yet occurred.

114.1.3 Train-Test Contamination

Information from test observations leaking into training, either directly or indirectly.

Common Sources of Contamination

Duplicate or near-duplicate observations: If the same patient appears in both train and test, the model may memorise rather than generalise. This is especially problematic in medical imaging where the same patient may have multiple scans.

Grouped observations split incorrectly: Multiple measurements from the same subject, location, or time window split across train/test. The model learns subject-specific patterns that will not generalise. For example, if predicting customer churn, putting some purchases from customer A in training and others in testing allows the model to “recognise” customer A in the test set.

Data augmentation after splitting: If augmented copies of test images are created before splitting, originals may land in training and augmented versions in testing—the model has effectively seen the test data.

114.1.4 Preprocessing Leakage

Fitting preprocessing steps on the entire dataset before splitting. This is perhaps the most subtle and commonly overlooked form of leakage.

Example: Standardisation

Consider standardising features to zero mean and unit variance:

$$x_{\text{scaled}} = \frac{x - \bar{x}}{\sigma_x}$$

If \bar{x} and σ_x are computed on the full dataset (including test data), then test observations influence the transformation applied to training data.

Correct procedure:

1. Split data into train/test
2. Compute \bar{x}_{train} and $\sigma_{x,\text{train}}$ from training data only
3. Apply this transformation to both train and test data

This also applies to: imputation (computing fill values), feature selection, PCA, and any data-driven transformation.

Why preprocessing leakage matters: The impact may seem minor—after all, how much can knowing the test set mean and variance really help? But consider:

- For small test sets, the statistics can be substantially influenced by test data
- More importantly, it sets a bad precedent and methodology that can lead to worse forms of leakage
- In some cases (e.g., feature selection), the leakage is severe

NB!

Feature selection before cross-validation is leakage. If you select the top 100 features by correlation with y using all data, then perform CV, your CV estimate is biased. The feature selection already “saw” the test folds.

Correct approach: Perform feature selection *inside* each CV fold. This means you select (potentially different) features for each fold, and the CV error honestly reflects the full feature-selection-plus-model-fitting pipeline.

114.2 Detecting Data Leakage

Prevention is ideal, but detection is also important for catching leakage that slipped through.

Leakage Detection Strategies

1. **Suspiciously good performance:** If your model achieves near-perfect accuracy on a hard problem, be suspicious. Real-world problems rarely have easy solutions.
2. **Production performance drop:** Large gap between offline metrics and online performance suggests leakage. The model looked great in testing but fails when deployed.
3. **Feature importance analysis:** If a feature is unexpectedly dominant, investigate whether it could encode the target. A feature you expected to be minor turning out to be decisive warrants scrutiny.
4. **Temporal validation:** For time-series problems, compare random CV to forward validation. Large discrepancy suggests temporal leakage.
5. **Ablation studies:** Remove suspicious features and check if performance degrades unreasonably. If removing a single feature causes accuracy to drop from 99% to 60%, that feature may be leaking information.

114.3 Prevention Framework

A systematic approach to preventing leakage before it occurs.

Leakage Prevention Checklist

Before training any model, answer these questions:

1. What is the prediction task in production?

- When will predictions be made?
- What information will be available at prediction time?
- Who will use the predictions and for what purpose?

2. Does my train/test split mirror production?

- Temporal ordering preserved? (If predicting the future, train on past only)
- Groups kept together? (All data from one patient in same split)
- No duplicate observations across splits?

3. Are all features legitimately available?

- Would each feature exist at prediction time?
- Is any feature derived from or caused by the target?
- Could any feature encode post-hoc information?

4. Is preprocessing done correctly?

- Fit preprocessing only on training data?
- Feature selection inside CV loops?
- No global statistics computed before splitting?

115 Sampling Schemes

Section Summary

The choice of sampling strategy depends on the data structure and the goal. **Simple random sampling** is the baseline—unbiased but potentially inefficient. **Stratified sampling** ensures representation of subgroups, critical for rare classes. **Cluster sampling** is practical when observations come in natural groups. **Systematic sampling** offers simplicity but risks aliasing with periodic patterns.

Before considering sophisticated adaptive sampling methods, we review the classical sampling schemes from survey methodology. These form the foundation upon which more advanced methods build.

115.1 Simple Random Sampling

Simple Random Sampling (SRS)

Each unit in the population has equal probability of selection:

$$P(\text{unit } i \text{ selected}) = \frac{n}{N}$$

where n is sample size and N is population size.

Properties:

- **Unbiased:** $\mathbb{E}[\bar{X}_{\text{sample}}] = \bar{X}_{\text{population}}$
- **Variance:** $\text{Var}(\bar{X}) = \frac{\sigma^2}{n} (1 - \frac{n}{N})$
- The factor $(1 - n/N)$ is the **finite population correction**

Unpacking the variance formula:

- σ^2/n is the familiar variance of the sample mean under infinite population sampling
- The factor $(1 - n/N)$ accounts for sampling *without replacement* from a finite population
- When $n \ll N$, this factor is approximately 1 and can be ignored
- When n is a substantial fraction of N , sampling without replacement reduces variance because we are “using up” population variability
- In the extreme case $n = N$, the variance is zero—we have sampled everyone

When to Use SRS

- Population is relatively homogeneous
- No important subgroups that might be undersampled by chance
- Complete sampling frame is available (a list of all population members)
- No structural dependencies (temporal, spatial, hierarchical)

SRS is the **gold standard** for unbiasedness but may be inefficient when population structure can be exploited.

115.2 Stratified Sampling

When the population has known subgroups with different characteristics, stratified sampling can improve efficiency.

Stratified Random Sampling

Divide the population into K non-overlapping strata, then sample independently within each stratum.

Let N_k be the population size and n_k the sample size for stratum k . Common allocation schemes:

Proportional allocation: $n_k \propto N_k$

$$n_k = n \cdot \frac{N_k}{N}$$

Each stratum is sampled in proportion to its population share.

Optimal (Neyman) allocation: $n_k \propto N_k \sigma_k$

$$n_k = n \cdot \frac{N_k \sigma_k}{\sum_{j=1}^K N_j \sigma_j}$$

where σ_k is the standard deviation within stratum k . This minimises variance of the population mean estimator.

Intuition for Neyman allocation: We should sample more from strata that are:

- **Larger** (N_k large): More of the population to learn about
- **More variable** (σ_k large): More uncertainty to reduce

A small, homogeneous stratum needs fewer samples; a large, heterogeneous stratum needs more.

Stratified Sampling: When and Why

Use when:

- Population has known subgroups with different characteristics
- Some subgroups are rare but important (class imbalance)
- You need guaranteed representation of all subgroups
- Within-stratum variance is lower than between-stratum variance

Variance reduction: If strata are internally homogeneous:

$$\text{Var}_{\text{stratified}} < \text{Var}_{\text{SRS}}$$

The more different the strata means, the greater the efficiency gain.

Example: Fraud Detection

In a credit card transaction dataset:

- 99.9% legitimate transactions
- 0.1% fraudulent transactions

With SRS of 1000 transactions, we expect ≈ 1 fraud case—insufficient to learn fraud patterns.

Stratified sampling with 500 from each class ensures adequate representation of fraud, enabling the model to learn the minority class.

Caveat: When deploying, predictions must account for the true class distribution. If we trained on 50/50 data, the model's probability estimates will be miscalibrated for the true 99.9/0.1 distribution. This is corrected via importance weighting (see below).

115.3 Cluster Sampling

Cluster Sampling

The population is divided into clusters (groups). A random sample of clusters is selected, and all units within selected clusters are observed.

Two-stage cluster sampling: Select clusters, then sample within clusters.

Design effect: The ratio of variance under cluster sampling to variance under SRS:

$$\text{DEFF} = 1 + (m - 1)\rho$$

where m is the average cluster size and ρ is the intra-cluster correlation. When $\rho > 0$ (units within clusters are similar), cluster sampling is less efficient than SRS.

Understanding the design effect:

- ρ measures how similar units within the same cluster are (0 = no more similar than random pairs; 1 = identical within clusters)
- If $\rho = 0$, clusters provide no information about similarity, and $\text{DEFF} = 1$ (cluster sampling is as efficient as SRS)
- If $\rho > 0$, observations within a cluster are redundant—you learn less per observation than with SRS
- Large clusters (m large) with high correlation (ρ large) lead to very inefficient sampling

Cluster Sampling: When and Why

Use when:

- No complete list of individuals, but lists of clusters exist (e.g., households, schools, hospitals)
- Cost of sampling is reduced by geographic clustering (cheaper to visit 10 schools than 100 students scattered across a city)
- Natural grouping structure exists

Tradeoff: Cluster sampling is often **administratively convenient** but **statistically inefficient** compared to SRS of the same size.

The key insight: clusters should be internally heterogeneous (each cluster “looks like” the population) for efficiency. Homogeneous clusters waste samples.

115.4 Systematic Sampling

Systematic Sampling

Select every k th unit from an ordered list, starting from a random position.

Procedure:

1. Compute sampling interval: $k = \lfloor N/n \rfloor$
2. Choose random start: $r \sim \text{Uniform}\{1, \dots, k\}$
3. Select units: $r, r + k, r + 2k, \dots$

Advantage: Simple to implement, spreads sample evenly across the list.

Disadvantage: If the list has periodic patterns with period k , systematic sampling can be severely biased.

NB!

Periodicity hazard: If a factory produces items on an assembly line and every 10th item comes from a particular machine, systematic sampling with $k = 10$ might sample entirely from one machine.

Always examine the ordering of the sampling frame for patterns before using systematic sampling. If in doubt, randomise the ordering first, or use simple random sampling.

115.5 Comparison of Sampling Schemes

Choosing a Sampling Scheme		
Method	Best when	Avoid when
Simple random	Homogeneous population	Rare subgroups matter
Stratified	Known important subgroups	Strata unknown or impractical
Cluster	Practical constraints on access	High intra-cluster correlation
Systematic	Need even spread; no periodicity	Periodic patterns in list

116 Importance Sampling

Section Summary
Importance sampling enables estimation of expectations under one distribution using samples from another. This is essential for correcting sampling bias and handling covariate shift —when training and test distributions differ. The key is appropriate reweighting, but beware: importance weights can have high variance, making estimates unstable.

116.1 The Core Idea

Suppose we want to compute an expectation under distribution p , but we can only sample from distribution q . This situation arises frequently:

- We oversampled rare events for training; now we want to evaluate on the true distribution
- Our training data comes from one population; we want to deploy to another
- The target distribution p is hard to sample from, but q is easy

Importance Sampling

We want to estimate $\mathbb{E}_p[f(X)]$ but can only sample from distribution q .

Key identity:

$$\begin{aligned}\mathbb{E}_p[f(X)] &= \int f(x)p(x)dx \\ &= \int f(x)\frac{p(x)}{q(x)}q(x)dx \\ &= \mathbb{E}_q\left[f(X)\frac{p(X)}{q(X)}\right]\end{aligned}$$

The **importance weights** are $w(x) = \frac{p(x)}{q(x)}$.

Estimator: Given samples $x_1, \dots, x_n \sim q$:

$$\hat{\mu}_{\text{IS}} = \frac{1}{n} \sum_{i=1}^n f(x_i)w(x_i) = \frac{1}{n} \sum_{i=1}^n f(x_i)\frac{p(x_i)}{q(x_i)}$$

Understanding the importance weights: The ratio $p(x)/q(x)$ tells us how much more (or less) likely observation x is under the target distribution p than under the sampling distribution q :

- If $p(x) > q(x)$: This region is underrepresented in our sample; upweight these observations
- If $p(x) < q(x)$: This region is overrepresented; downweight these observations
- If $p(x) = q(x)$ everywhere: The distributions match; weights are all 1

The Unifying Principle

Multiply by what you want, divide by what you have.

This captures the essence of importance sampling: we have samples from q (“what we have”) and want expectations under p (“what we want”). The importance weight p/q corrects for this mismatch.

Self-Normalised Importance Sampling

When p and q are known only up to normalising constants (common in Bayesian inference), we cannot compute $w(x) = p(x)/q(x)$ exactly.

The **self-normalised** estimator uses:

$$\hat{\mu}_{\text{SNIS}} = \frac{\sum_{i=1}^n f(x_i)w(x_i)}{\sum_{i=1}^n w(x_i)}$$

where $w(x_i) = \tilde{p}(x_i)/\tilde{q}(x_i)$ uses unnormalised densities \tilde{p} and \tilde{q} .

This is biased but consistent, and often has lower variance than the standard IS estimator because the normalisation constants cancel.

116.2 Variance Considerations

Importance sampling is unbiased, but it can have very high variance—sometimes so high that the estimator is practically useless.

Variance of Importance Sampling Estimator

$$\text{Var}_q(\hat{\mu}_{\text{IS}}) = \frac{1}{n} \text{Var}_q(f(X)w(X)) = \frac{1}{n} (\mathbb{E}_q[f(X)^2 w(X)^2] - \mu^2)$$

The variance depends on $\mathbb{E}_q[w(X)^2]$ —the second moment of the importance weights.

Effective sample size (ESS) quantifies efficiency:

$$\text{ESS} = \frac{(\sum_i w_i)^2}{\sum_i w_i^2} = \frac{n}{1 + \text{Var}(w_i)/\bar{w}^2}$$

When weights are uniform, $\text{ESS} = n$. When weights are highly variable, $\text{ESS} \ll n$.

Interpreting effective sample size: ESS answers the question: “How many i.i.d. samples from p would give us the same precision as our n importance-weighted samples from q ?” If $\text{ESS} = 10$ but $n = 1000$, we are effectively throwing away 99% of our samples due to weight variability.

NB!

Weight degeneracy: If p and q have poor overlap, a few observations may receive almost all the weight. This leads to:

- Extremely high variance estimates
- Effective sample size approaching 1
- Estimates dominated by a single observation

Rule of thumb: Importance sampling works well when p/q is bounded. If p has heavier tails than q , expect trouble—the regions where p is large but q is small will produce enormous weights.

116.3 Application: Covariate Shift

One of the most important applications of importance sampling in machine learning is correcting for distribution shift between training and deployment.

Covariate Shift

Training data comes from distribution $p_{\text{train}}(x, y) = p(y|x)p_{\text{train}}(x)$.

Test data comes from distribution $p_{\text{test}}(x, y) = p(y|x)p_{\text{test}}(x)$.

The conditional $p(y|x)$ is the same, but the marginal over x differs.

Examples:

- Training on hospital A, deploying at hospital B (different patient demographics)
- Training on summer data, predicting in winter
- Training on volunteer survey responses, inferring for general population
- Training on English Wikipedia, deploying on legal documents

Why covariate shift matters: If the feature distribution changes but the relationship between features and outcomes does not, we can still use our model—but we need to account for the changed distribution when evaluating performance.

Correcting for Covariate Shift

The population risk under test distribution:

$$R_{\text{test}}(f) = \mathbb{E}_{p_{\text{test}}(x)}[\ell(y, f(x))]$$

Using importance weighting with training samples:

$$\hat{R}_{\text{test}}(f) = \frac{1}{n} \sum_{i=1}^n \frac{p_{\text{test}}(x_i)}{p_{\text{train}}(x_i)} \ell(y_i, f(x_i))$$

Challenge: We rarely know p_{train} and p_{test} exactly. Common approaches:

- **Density ratio estimation:** Estimate the ratio $p_{\text{test}}/p_{\text{train}}$ directly using kernel methods or neural networks
- **Domain adaptation:** Learn representations that are invariant to the domain
- **Discriminative approach:** Train a classifier to distinguish train from test; use predicted probabilities to construct weights

Key Applications of Importance Sampling

Importance sampling is a general tool for:

- **Correcting non-random sampling:** Oversample rare events for training, then downweight for evaluation
- **Adapting to distribution shift:** Reweight training data to match deployment distribution
- **Variance reduction:** Sample where the integrand is large (in Monte Carlo integration)
- **Bayesian computation:** Sample from tractable proposal distributions to approximate intractable posteriors

117 Random vs Non-Random Sampling

Why Random Sampling?

We care about **population risk**:

$$R(f) = \mathbb{E}_{p(x,y)}[\ell(y, f(x))]$$

This measures how well our model f performs on average across all possible inputs, weighted by how likely each input is to occur.

Random sampling ensures ERM approximates population risk without bias.

Population risk is a theoretical quantity we cannot compute directly (we do not have access to

the entire population). We approximate it via ERM on a sample:

$$\hat{R}(f) = \frac{1}{n} \sum_{i=1}^n \ell(y_i, f(x_i))$$

When the sample is drawn uniformly at random, $\hat{R}(f)$ is an unbiased estimator of $R(f)$. But when is it worth departing from random sampling?

117.1 Heteroskedastic Noise

Heteroskedasticity

Definition: The variance of the error term depends on the predictor values.

$$y_i = X_i\beta + \epsilon_i, \quad \epsilon_i \sim \mathcal{N}(0, \sigma^2(X_i))$$

where $\sigma^2(X_i)$ varies with X_i , rather than being constant.

Contrast with homoskedasticity: Under homoskedasticity, $\sigma^2(X_i) = \sigma^2$ for all i —the noise is the same everywhere.

Why heteroskedasticity matters for sampling: If noise varies across the feature space, then:

- Some regions are inherently harder to predict (high noise)
- Some regions are easier (low noise)
- Uniform sampling may waste effort on easy regions while undersampling hard ones

Examples of Heteroskedasticity

Income vs expenditure: Variance of expenditure increases with income. High earners have more discretionary spending variability—a millionaire might spend \$100 or \$10,000 on a dinner, while a minimum-wage worker's dinner expenses are more constrained.

Company size vs returns: Larger companies tend to have less variable returns than smaller companies (large companies are more stable).

Measurement precision: Some regions of feature space may be harder to measure precisely. For example, measuring very small quantities (near detection limits) or extreme temperatures introduces more noise.

Learning curves: Early in training, predictions are more variable than after convergence.

117.1.1 Detecting Heteroskedasticity

Before addressing heteroskedasticity, we need to detect it.

Detection Methods

Visual inspection: Plot residuals $e_i = y_i - \hat{y}_i$ against fitted values \hat{y}_i or predictors X_i . Look for “funnel” or “fan” patterns where residual spread changes systematically.

Breusch-Pagan test: Regress squared residuals on predictors. Test whether coefficients are jointly zero.

$$e_i^2 = \gamma_0 + \gamma_1 X_{i1} + \cdots + \gamma_p X_{ip} + \nu_i$$

Under H_0 (homoskedasticity), $nR^2 \sim \chi_p^2$. A significant test suggests heteroskedasticity.

White test: Similar to Breusch-Pagan but includes squares and cross-products of predictors. More general but uses more degrees of freedom.

117.1.2 Consequences for OLS

What happens if we ignore heteroskedasticity and use ordinary least squares?

OLS under Heteroskedasticity

Recall the OLS estimator: $\hat{\beta} = (X^\top X)^{-1} X^\top y$

Under heteroskedasticity:

- $\hat{\beta}$ is still **unbiased**: $\mathbb{E}[\hat{\beta}] = \beta$
- $\hat{\beta}$ is still **consistent**: $\hat{\beta} \xrightarrow{P} \beta$
- But $\hat{\beta}$ is **no longer efficient**: there exist estimators with lower variance
- Standard errors are **wrong**: the usual formula $\hat{\sigma}^2(X^\top X)^{-1}$ assumes homoskedasticity

The consequence: hypothesis tests and confidence intervals based on standard OLS standard errors are invalid. You may reject the null hypothesis too often (if you underestimate standard errors) or too rarely (if you overestimate them).

117.1.3 Solution 1: Weighted Least Squares

If we know (or can estimate) the variance structure, we can incorporate it directly into estimation.

Weighted Least Squares (WLS)

If we know (or can estimate) the variance function $\sigma^2(X_i) = \sigma_i^2$, use weights $w_i = 1/\sigma_i^2$.

Objective:

$$\hat{\beta}_{\text{WLS}} = \arg \min_{\beta} \sum_{i=1}^n w_i(y_i - X_i\beta)^2$$

Solution:

$$\hat{\beta}_{\text{WLS}} = (X^\top W X)^{-1} X^\top W y$$

where $W = \text{diag}(w_1, \dots, w_n)$.

Interpretation: Observations with high variance (unreliable) get low weight; observations with low variance (precise) get high weight.

Intuition for WLS: Think of it as giving each observation a “credibility score” based on how noisy it is. High-noise observations are less trustworthy and should influence the fit less. This is similar to how a meta-analysis weights studies by their precision.

WLS Properties

When weights correctly specify relative variances:

- $\hat{\beta}_{\text{WLS}}$ is unbiased
- $\hat{\beta}_{\text{WLS}}$ is the **Best Linear Unbiased Estimator** (BLUE)
- $\hat{\beta}_{\text{WLS}}$ achieves the Gauss-Markov bound
- Standard inference (t-tests, F-tests) is valid

In practice, weights are often estimated, introducing additional uncertainty not accounted for in standard WLS inference. This is sometimes called “feasible GLS” and requires more careful inference.

117.1.4 Solution 2: Robust Standard Errors

An alternative approach: do not try to model the variance structure; just make inference robust to it.

Heteroskedasticity-Robust Standard Errors

Rather than trying to model the variance structure, use **robust** (Huber-White, sandwich) standard errors.

The robust covariance estimator:

$$\widehat{\text{Var}}(\hat{\beta}) = (X^\top X)^{-1} \left(\sum_{i=1}^n e_i^2 x_i x_i^\top \right) (X^\top X)^{-1}$$

where $e_i = y_i - X_i \hat{\beta}$ are the OLS residuals.

This is valid under heteroskedasticity without specifying the form of $\sigma^2(X)$. The “sandwich” name comes from the structure: bread-meat-bread, where the meat is the middle term that accounts for heteroskedasticity.

When to Use Which

Use WLS when:

- You have a good model for the variance structure
- Efficiency gains are important (small samples)
- The goal is prediction (WLS predictions are optimal)

Use robust SEs when:

- Variance structure is unknown or complex
- Primary goal is valid inference, not efficiency
- Sample size is large (robust SEs need asymptotics)

Best practice: Report robust SEs by default. They are (almost) never wrong; classical SEs can be badly wrong if heteroskedasticity is present.

117.2 Implications for Sampling

Heteroskedasticity motivates departing from uniform random sampling.

Non-Uniform Sampling Motivation

High-noise regions require more samples to achieve the same accuracy. This motivates **non-uniform sampling**:

- Sample more densely where the signal-to-noise ratio is low
- Sample less where predictions are already precise

But care is needed: non-uniform sampling for training can be corrected via reweighting; non-uniform sampling for testing is more dangerous.

NB!

Non-random sampling for **training** data can be corrected via importance weighting.

Non-random sampling for **test** data is dangerous—you may have no idea how the model performs on the true population. Your test set may look good simply because it avoids the hard cases. There is no way to correct for not knowing what you do not know.

118 Active Learning

Section Summary

Active learning selects the most informative points for labelling, reducing annotation costs while maintaining model quality. **Uncertainty sampling** queries where the model is least confident. **Query-by-committee** queries where an ensemble disagrees. **BALD** targets reducible (epistemic) uncertainty, not inherent noise. Active learning helps most when the decision boundary is complex and labels are expensive.

Active learning optimises the training process by iteratively selecting the most informative data points for labelling. This is particularly valuable when labelling is expensive or time-consuming—medical diagnoses, expert annotations, physical experiments.

Active Learning: The Setup

Pool-based active learning:

1. Start with a large pool of unlabelled data $\mathcal{U} = \{x_1, \dots, x_N\}$
2. Maintain a (initially small) labelled set \mathcal{L}
3. Iteratively:
 - (a) Train model on \mathcal{L}
 - (b) Use acquisition function to select most informative $x \in \mathcal{U}$
 - (c) Query oracle for label; add (x, y) to \mathcal{L}
 - (d) Remove x from \mathcal{U}

Stream-based active learning: Observations arrive one at a time; decide immediately whether to query.

The key question: What makes a point “informative”?

118.1 Criteria for Selecting Data Points

Several strategies exist for identifying informative points:

- **Uncertainty sampling:** Choose points where the model is most uncertain
- **Query by committee:** Maintain multiple models and choose points where they disagree most
- **Expected model change:** Select points that, when labelled, would cause the largest change to model parameters

- **Expected error reduction:** Choose points expected to most reduce overall error on the unlabelled set
- **Density-weighted methods:** Combine uncertainty with density, preferring uncertain points that are also representative

118.2 Uncertainty Sampling

The simplest and most common active learning approach: select points where the model is most uncertain about its prediction.

Uncertainty Measures (Classification)

For a classifier producing class probabilities $p_c = P(Y = c|x)$:

Least confidence:

$$u_{LC}(x) = 1 - \max_c p_c$$

High when the model is unsure about its best guess. Simplest measure; focuses only on the top prediction. Best when boosting confidence in top predictions is the priority.

Margin sampling:

$$u_{\text{margin}}(x) = p_{\text{top}} - p_{\text{second}}$$

Low when the model cannot distinguish the top two classes. (Note: select where margin is *smallest*.) Useful for refining decision boundaries between classes.

Entropy:

$$u_{\text{entropy}}(x) = - \sum_c p_c \log p_c = H(Y|x)$$

High when predictions are spread across classes; low when confident. Considers the full probability distribution. Best when any misclassification is equally costly.

For binary classification, all three are equivalent (monotonic transformations of each other).

Intuition: Points near the decision boundary are most informative because:

- The model is most uncertain about them
- They help refine exactly where the boundary should be
- A small change in the model could flip their predictions

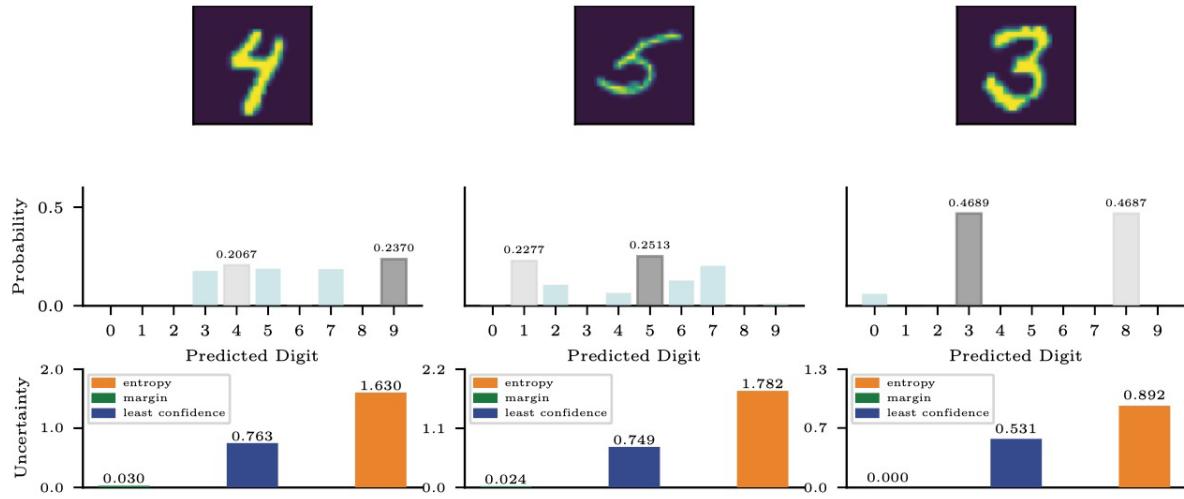


Figure 67: Uncertainty sampling in action: points near the decision boundary (where the model is uncertain) are selected for labelling. These are precisely the points where additional labels provide the most information for refining the boundary.

Uncertainty Sampling for Regression

For regression with predictive distribution $p(y|x)$:

Predictive variance:

$$u(x) = \text{Var}[y|x]$$

For linear regression: $\text{Var}[y|x] = \sigma^2(1 + x^\top(X^\top X)^{-1}x)$

The term $x^\top(X^\top X)^{-1}x$ is the **leverage** of point x . Points with high leverage (far from training data in feature space) have high predictive variance.

118.3 Query-by-Committee

A more robust approach that uses an ensemble of models.

Query-by-Committee (QBC)

Maintain a **committee** of models $\{f_1, \dots, f_C\}$, each trained on the same data but with different:

- Random initialisations (neural networks)
- Bootstrap samples (bagging)
- Hyperparameters
- Posterior samples (Bayesian)

Query points where committee members **disagree**:

$$u_{\text{QBC}}(x) = \text{disagreement}(f_1(x), \dots, f_C(x))$$

Disagreement measures:

- **Vote entropy**: Entropy of the vote distribution
- **KL divergence**: Average divergence from consensus
- **Variance**: For regression, variance across committee predictions

QBC vs Uncertainty Sampling

QBC advantages:

- Naturally captures model uncertainty, not just predictive uncertainty
- Robust to overconfident models (a single model may be wrongly confident)
- Can distinguish “hard examples” from “ambiguous examples”

QBC disadvantages:

- Computationally expensive (train and query multiple models)
- Committee construction is non-trivial
- May select points where *all* models are wrong but confident

118.4 Expected Model Change

Rather than focusing on uncertainty, focus on how much labelling a point would change the model.

Expected Model Change (EMC)

Select points that would **most change the model** if labelled.

$$u_{\text{EMC}}(x) = \mathbb{E}_{y|x} [\|\nabla_{\theta} \ell(y, f_{\theta}(x))\|]$$

Intuition: Points with large expected gradient would cause large parameter updates, suggesting they contain information not yet captured by the model.

Variants:

- **Expected gradient length (EGL):** As above
- **Expected model change:** Change in predictions across the input space
- **Fisher information:** Expected gradient outer product

Computationally expensive: requires computing gradients for each candidate point under each possible label.

118.5 Bayesian Active Learning by Disagreement (BALD)

A more sophisticated approach that addresses a key limitation of simple uncertainty sampling: not all uncertainty is reducible.

The Problem with Pure Uncertainty Sampling

Consider a region where the true relationship is genuinely noisy—even with infinite data, predictions would remain uncertain. Pure uncertainty sampling might waste labelling budget on these inherently unpredictable regions.

BALD addresses this by distinguishing between:

- **Epistemic uncertainty:** Uncertainty about the model parameters, which *can* be reduced with more data
- **Aleatoric uncertainty:** Inherent noise in the data, which *cannot* be reduced

BALD: Formal Definition

The **key insight**: Not all uncertainty is equal. Decompose total uncertainty:

$$\underbrace{H(Y|x, \mathcal{D})}_{\text{Total uncertainty}} = \underbrace{\mathbb{E}_{p(\theta|\mathcal{D})}[H(Y|x, \theta)]}_{\text{Aleatoric (irreducible)}} + \underbrace{I(Y; \theta|x, \mathcal{D})}_{\text{Epistemic (reducible)}}$$

Aleatoric uncertainty: Inherent randomness in Y given x . Cannot be reduced by more data—some x values are just noisy.

Epistemic uncertainty: Uncertainty due to not knowing the true model. Can be reduced by more data.

BALD acquisition function: Select points maximising epistemic uncertainty:

$$u_{\text{BALD}}(x) = H(Y|x, \mathcal{D}) - \mathbb{E}_{p(\theta|\mathcal{D})}[H(Y|x, \theta)]$$

This is the mutual information between Y and θ given x .

Unpacking the BALD formula:

- $H(Y|x, \mathcal{D})$: Total uncertainty in predicting y given x and all observed data. This is what uncertainty sampling uses.
- $\mathbb{E}_{p(\theta|\mathcal{D})}[H(Y|x, \theta)]$: Average uncertainty when we know the true parameters. This is the inherent noise we cannot reduce, averaged over our posterior belief about θ .
- The difference: Epistemic uncertainty—the part we *can* reduce by collecting more data.

BALD: Why It Matters

Consider two uncertain predictions:

1. A fair coin flip: $P(Y = 1|x) = 0.5$ with high confidence
2. Model uncertainty: Different plausible models give $P(Y = 1|x) \in [0.2, 0.8]$

Both have high entropy, but:

- Case 1: Labelling will not help—the example is inherently noisy. All models agree it is a coin flip.
- Case 2: Labelling will help distinguish between models. The models disagree about what the probability should be.

BALD correctly targets case 2; naive uncertainty sampling treats them equally.

BALD via Information Gain

Equivalently, BALD selects points that maximise the expected information gain about model parameters:

$$\mathbb{E}_{y|x,\mathcal{D}} [D_{KL} [p(\theta|\mathcal{D}, x, y) \| p(\theta|\mathcal{D})]]$$

Breaking this down:

- $p(\theta|\mathcal{D})$: Current posterior over model parameters given observed data \mathcal{D}
- $p(\theta|\mathcal{D}, x, y)$: Updated posterior after observing a new point (x, y)
- $D_{KL}[\cdot\|\cdot]$: Kullback-Leibler divergence measuring how much the posterior would change
- The expectation is taken over possible values of y (since we do not know y yet)

Intuitively: we “hallucinate” what might happen if we labelled point x , and measure how much we would learn about the model.

118.6 When Does Active Learning Help?

Active learning is not always beneficial. Understanding when it helps guides its application.

Conditions for Active Learning Success

Active learning provides the largest gains when:

1. **Labels are expensive**: Medical diagnoses, expert annotations, physical experiments
2. **Unlabelled data is abundant**: Web data, sensor readings, images
3. **Decision boundary is complex**: Many regions of uncertainty exist
4. **Initial model is reasonable**: If the model is completely wrong, uncertainty sampling may focus on the wrong regions

Active learning may provide **little benefit** when:

- Labels are cheap relative to unlabelled data collection
- Data is uniformly informative (no regions are more important)
- Class imbalance is extreme (random sampling rarely finds minority class)

NB!

Cold start problem: Active learning needs a reasonable initial model to identify informative points. With very few initial labels, the model may be so poor that its uncertainty estimates are meaningless.

Mitigation: Start with random sampling until the model achieves baseline competence, then switch to active selection.

119 Correcting for Non-Uniform Sampling

When we sample non-uniformly for training data (as in active learning), we need to correct the resulting bias to recover unbiased estimates of population risk.

119.1 The Problem

Suppose we sample observation i with probability p_i (which may depend on X_i). Our sample no longer represents the true population distribution. If we naively apply ERM, we get a biased estimate of population risk.

119.2 Inverse Probability Weighting (IPW)

Inverse Probability Weighting

To recover an unbiased estimate of population risk from a non-uniform sample, reweight each observation by the inverse of its sampling probability:

$$\hat{R}(f) = \frac{1}{\sum_i 1/p_i} \sum_{i=1}^n \frac{1}{p_i} \ell(y_i, f(x_i))$$

Points sampled with low probability get high weight; points sampled with high probability get low weight. This “undoes” the sampling bias.

Unbiasedness: If $p_i > 0$ for all i in the population:

$$\mathbb{E}[\hat{R}(f)] = R(f)$$

Why IPW works: Let $s_i \in \{0, 1\}$ indicate whether observation i was sampled. The key insight is:

$$\mathbb{E}\left[\frac{s_i}{p_i} \cdot \ell(y_i, f(x_i))\right] = \mathbb{E}[\ell(y_i, f(x_i))]$$

since $\mathbb{E}[s_i] = p_i$. The inverse weighting exactly cancels the sampling probability.

Practical Implications of IPW

IPW allows us to:

- Use non-uniform sampling strategies (active learning, leverage score sampling) to collect training data efficiently
- Then correct for the induced bias to get unbiased estimates
- Achieve both **efficiency** (fewer labels needed) and **unbiasedness** (correct population risk estimates)

NB!

IPW requires that $p_i > 0$ for all observations you want to generalise to (**positivity assumption**). If some regions of the population have $p_i = 0$, no amount of reweighting can tell you about them.

Also beware of **extreme weights**: if some p_i are very small, those observations receive very large weights, leading to high variance estimates. This is the same weight degeneracy problem as in importance sampling.

120 Leverage Score Sampling

Section Summary

Leverage scores measure how influential each observation is for linear regression. High-leverage points lie far from the centre of the predictor space and disproportionately affect the fit. Sampling proportionally to leverage scores enables efficient approximation of full-data regression with sample size depending on p (features) rather than n (observations).

For ordinary least squares (OLS) regression, some observations are inherently more influential than others. Leverage score sampling exploits this structure.

120.1 Leverage Scores in Linear Regression

Leverage Scores

For linear regression with design matrix X ($n \times p$), the **leverage** of observation i is:

$$h_{ii} = x_i^\top (X^\top X)^{-1} x_i$$

This is the i th diagonal element of the **hat matrix**:

$$H = X(X^\top X)^{-1} X^\top$$

The hat matrix “puts the hat on y ”: $\hat{y} = Hy$.

Why “hat matrix”?: The matrix H transforms the observed responses y into the fitted values \hat{y} . We say it “puts the hat on y ” because it takes y and produces \hat{y} .

Properties of Leverage Scores

1. **Range**: $0 \leq h_{ii} \leq 1$ (with intercept: $1/n \leq h_{ii} \leq 1$)
2. **Sum**: $\sum_{i=1}^n h_{ii} = \text{tr}(H) = p$ (number of parameters)
3. **Average**: $\bar{h} = p/n$
4. **Self-influence**: $\frac{\partial \hat{y}_i}{\partial y_i} = h_{ii}$ —leverage measures how much changing y_i affects its own prediction

Understanding property 4: The leverage h_{ii} tells us: if we change y_i by one unit, how much does \hat{y}_i change? High-leverage observations have more influence on their own fitted values—and on the regression line overall.

120.2 Geometric Interpretation

Leverage as Distance from Centre

Leverage measures how “unusual” an observation’s predictor values are.

For centred data with $X^\top X = n\hat{\Sigma}$ (sample covariance):

$$h_{ii} = \frac{1}{n}(x_i - \bar{x})^\top \hat{\Sigma}^{-1}(x_i - \bar{x}) + \frac{1}{n}$$

The first term is the **Mahalanobis distance** from the centroid. Points far from the centre (in the metric defined by the covariance) have high leverage.

Intuition: Imagine fitting a regression line through a cloud of points. Points near the centre of the cloud have many neighbours that “anchor” them—moving one such point does not change the line much. But an isolated point at the edge of the cloud has no neighbours to counterbalance it—it can “tilt” the entire regression line.

Why Leverage Matters

High-leverage observations:

- Have more influence on the fitted coefficients
- Determine the regression line more strongly
- If they also have unusual y values, they are **influential** (change predictions substantially)
- Are candidates for outlier investigation

Connection to Week 4: The LOOCV shortcut uses leverage:

$$e_i^{(-i)} = \frac{e_i}{1 - h_{ii}}$$

where $e_i^{(-i)}$ is the leave-one-out residual and e_i is the ordinary residual. High-leverage points have their residuals inflated more when computing LOOCV.

120.3 Leverage Score Sampling for Large-Scale Regression

Leverage Score Sampling

For very large n , computing the full OLS solution is expensive ($O(np^2)$). Instead:

1. Compute (approximate) leverage scores h_{ii} for all observations
2. Sample $m \ll n$ observations with probability $\propto h_{ii}$
3. Fit OLS on the subsample (with importance weights)

Theoretical guarantee (Cohen & Peng, 2014): With $m = O(p \log n / \epsilon^2)$ samples, the subsampled solution achieves:

$$\|X\hat{\beta}_{\text{sub}} - X\hat{\beta}\|_2 \leq (1 + \epsilon)\|X\hat{\beta} - y\|_2$$

The sample size depends on p (features) not n (population size)!

The remarkable implication: This result says we can achieve near-optimal regression accuracy using a number of samples that scales with the complexity of the model (p), not the size of the data (n). For $n = 10^9$ and $p = 100$, we might need only thousands of samples instead of billions.

Why Leverage Sampling Works

Intuition: Regression is most sensitive to observations that “span” the feature space. High-leverage points define the extremes; middle points are redundant.

Random sampling failure: Random sampling might miss high-leverage points entirely, leading to a poorly conditioned subsample. If all sampled points are near the centre, we cannot reliably estimate the slope.

Leverage sampling: Ensures high-leverage points are included, preserving the essential geometry of the problem.

120.4 Connection to Optimal Experimental Design

Leverage score sampling connects to a classical statistical problem: experimental design.

Optimal Experimental Design

In experimental design, we choose which x values to observe to minimise estimation uncertainty.

D-optimal design: Maximise $\det(X^\top X)$, equivalently minimise volume of confidence ellipsoid for β .

A-optimal design: Minimise $\text{tr}((X^\top X)^{-1})$, the average variance of coefficient estimates.

Connection: D-optimal designs tend to place observations at high-leverage positions—the “corners” of the design space. If you can choose where to collect data, put observations at the extremes.

Leverage score sampling is an adaptive version: given existing data, sample where additional observations would be most informative.

121 Random Fourier Features

Kernel methods are powerful but computationally expensive. Random Fourier Features provide an efficient approximation.

121.1 The Computational Challenge

The Gram Matrix Problem

Kernel methods transform input data into a higher-dimensional space where linear methods become more powerful. The kernel trick avoids explicitly computing high-dimensional features, but requires computing the Gram matrix K , where $K_{ij} = k(x_i, x_j)$.

Computational complexity:

- Storage: $O(n^2)$ for the $n \times n$ matrix
- Computation: Up to $O(n^3)$ for inversion or eigendecomposition

For large n , this becomes prohibitive. With $n = 10^6$, storing the Gram matrix requires ~ 8 TB (at 8 bytes per entry)!

121.2 The Random Fourier Features Approximation

Random Fourier Features (RFF)

For shift-invariant kernels (e.g., the RBF/Gaussian kernel), the kernel function can be approximated as:

$$k(x, x') \approx \frac{2}{R} \sum_{r=1}^R \cos(w_r^\top x + b_r) \cos(w_r^\top x' + b_r)$$

where:

- $w_r \sim \mathcal{N}(0, I/\sigma^2)$: Random frequency vectors drawn from a Gaussian
- $b_r \sim \text{Uniform}(0, 2\pi)$: Random phase shifts
- R : Number of random features (a hyperparameter)
- σ : Kernel bandwidth parameter

Theoretical basis: Bochner's theorem states that any positive-definite shift-invariant kernel is the Fourier transform of a probability measure. By sampling from this distribution, we create explicit random features that approximate the kernel.

Unpacking Bochner's theorem: A shift-invariant kernel depends only on $x - x'$: $k(x, x') = k(x - x')$. Bochner's theorem says such kernels can be written as:

$$k(x - x') = \int e^{iw^\top(x-x')} p(w) dw$$

for some probability distribution $p(w)$. By sampling $w \sim p$ and using the real part (cosines), we approximate this integral.

RFF Computational Savings

Original kernel regression: $O(n^3)$ complexity

With RFF: $O(R^3)$ complexity, where $R \ll n$

We transform each input x into a new feature vector $\phi(x) \in \mathbb{R}^R$:

$$\phi(x) = \sqrt{\frac{2}{R}} \begin{bmatrix} \cos(w_1^\top x + b_1) \\ \vdots \\ \cos(w_R^\top x + b_R) \end{bmatrix}$$

Then run standard linear regression on these transformed features. This reduces complexity dramatically while maintaining good approximation quality.

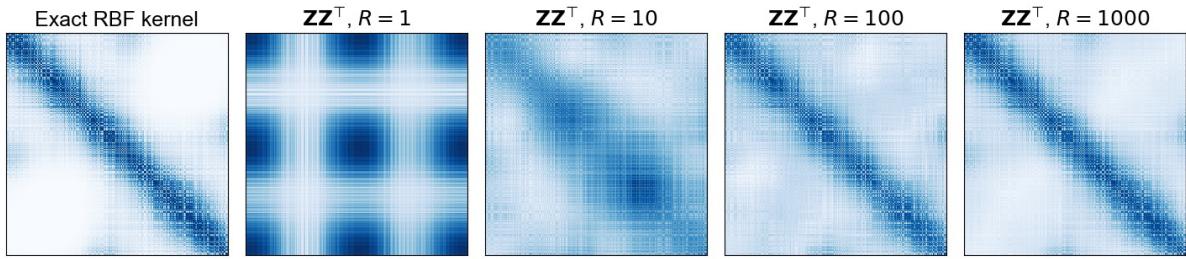


Figure 68: RFF approximation quality improves with more random features. Panel 1 shows the true RBF kernel function we aim to approximate. Subsequent panels show how the approximation improves as we increase the number of random features R .

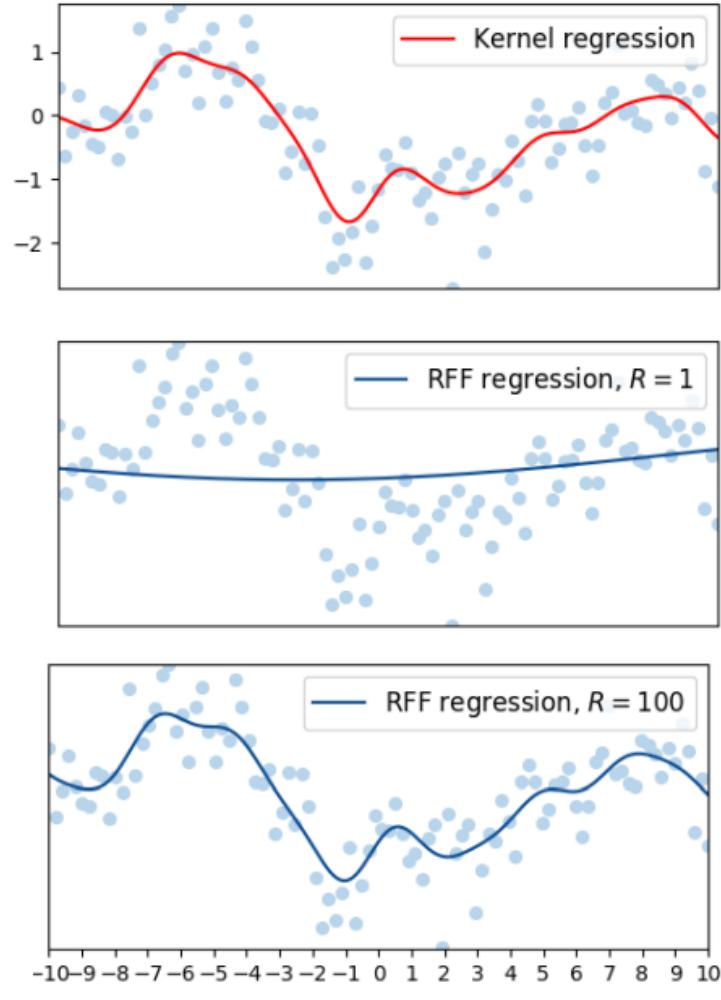


Figure 69: Regression predictions using Random Fourier Features. The approximation quality depends on choosing a sufficient number of random features R relative to the complexity of the underlying function.

Combining Leverage Scores with RFF

A powerful combination: first use RFF to create explicit features $\phi(x)$, then compute leverage scores in this transformed space. This allows efficient kernel regression even on massive datasets:

1. Transform all points using RFF: $\phi(x_i) = \sqrt{2/R}[\cos(w_1^\top x_i + b_1), \dots, \cos(w_R^\top x_i + b_R)]^\top$
2. Compute leverage scores in the RFF feature space
3. Sample according to leverage scores
4. Run regression on the sampled points

This combines the approximation power of kernels with the sampling efficiency of leverage scores.

122 Practical Sampling Pipelines

Section Summary

Proper data splitting is fundamental to reliable ML. The basic train/validation/test split serves different purposes: training for fitting, validation for tuning, testing for final evaluation. For structured data, splits must respect the structure—groups stay together, time moves forward. **Never look at the test set until the very end.**

122.1 Train/Validation/Test Splits

The Three-Way Split

Training set (~60-80%): Fit model parameters.

Validation set (~10-20%): Tune hyperparameters, select among models.

Test set (~10-20%): Final, unbiased performance estimate.

Why three sets?

- Training error is optimistically biased (model has seen the data)
- Validation error becomes biased after hyperparameter tuning (information leakage)
- Test error on a truly held-out set provides honest assessment

NB!

Test set discipline: The test set must be used **exactly once**, at the very end. If you tune hyperparameters based on test performance, the test set becomes another validation set, and you have no honest estimate of generalisation.

Kaggle competitions enforce this: you submit predictions, and the test labels are never revealed until the competition ends.

122.2 Time Series: Forward Validation

Time Series Splits

For temporal data, the future cannot be used to predict the past.

Walk-forward validation:

1. Train on $[1, t]$
2. Validate on $[t + 1, t + h]$ (horizon h)
3. Expand: Train on $[1, t + h]$
4. Validate on $[t + h + 1, t + 2h]$
5. Repeat

Sliding window: Use fixed-size training window instead of expanding.

Gap: Sometimes include a gap between train and test to avoid information leakage from autocorrelation.

Time Series Best Practices

1. Never shuffle time series data
2. Training set must precede validation/test temporally
3. Be cautious with features that could leak future information
4. Consider multiple test periods to assess stability
5. Account for seasonality (test should cover same seasons as deployment)

See Week 4 for detailed treatment of time series cross-validation.

122.3 Cross-Validation Revisited

When to Use Which CV Strategy

Standard K-fold: i.i.d. data, sufficient sample size.

Stratified K-fold: Classification with imbalanced classes. Ensures each fold has representative class distribution.

Group K-fold: Data has natural clusters (patients, users, locations). Keep all observations from a group in the same fold.

Time series split: Temporal ordering matters. Train on past, test on future.

Nested CV: Need both hyperparameter tuning and honest performance estimate. Outer loop for evaluation, inner loop for tuning.

For detailed coverage of cross-validation methods, see Week 4.

123 Multi-Armed Bandits

Section Summary

Multi-armed bandits address the exploration-exploitation tradeoff in sequential decision-making. ϵ -greedy is simple but wasteful. UCB provides principled optimism under uncertainty with regret guarantees. Thompson sampling takes a Bayesian approach, sampling from posterior beliefs. All achieve sublinear regret, meaning cumulative loss relative to the best arm grows slower than linearly.

We now shift from sampling for better models to sampling for better **decisions**. When the goal is maximising reward rather than building an accurate model, bandit algorithms provide the appropriate framework.

Multi-Armed Bandit Problem

Imagine a row of slot machines (“one-armed bandits”), each with a different (unknown) payout rate. You have a limited number of plays. How should you allocate your plays to maximise total winnings?

Formally, at each time t :

1. Choose action a_t from K available options (“arms”)
2. Observe reward r_t (stochastic, depending on which arm was pulled)

Objective: Maximise cumulative reward $\sum_{t=1}^T r_t$

Contextual bandits: Reward depends on context x_t : $r_t = r(a_t, x_t)$. This is more general—the best action may depend on the situation.

Why “bandit”?: The metaphor comes from being “robbed” by slot machines. A multi-armed bandit is a collection of slot machines, each with a hidden probability of paying out.

Practical applications:

- **A/B testing**: Which website design leads to more clicks?
- **Clinical trials**: Which treatment is most effective?
- **Advertisement placement**: Which ad generates most revenue?
- **Recommendation systems**: Which item should we recommend?

123.1 Exploration vs Exploitation

The Core Tradeoff

Exploitation: Choose the action with highest estimated reward. Make the best decision given what you know now.

Exploration: Try actions with uncertain rewards to learn more. Gather information that might lead to better decisions later.

The tension:

- Too much exploitation \Rightarrow miss better options you never tried
- Too much exploration \Rightarrow waste resources on actions you already know are bad

A good bandit algorithm must balance these competing goals dynamically over time—exploring early and exploiting later.

Regret

Regret measures how much worse we do compared to always playing the best arm:

$$\text{Regret}_T = T \cdot \mu^* - \sum_{t=1}^T \mu_{a_t}$$

where $\mu^* = \max_a \mu_a$ is the expected reward of the best arm, and μ_{a_t} is the expected reward of the arm we chose at time t .

A good algorithm achieves **sublinear regret**: $\text{Regret}_T = o(T)$. This means the average regret per step goes to zero: we make fewer and fewer mistakes over time.

Why sublinear regret matters:

- **Linear regret $O(T)$:** Average performance never improves—each mistake is equally costly. This happens with pure random exploration.
- **Sublinear regret $O(\sqrt{T \log T})$:** We learn! Per-round regret $\rightarrow 0$ as $T \rightarrow \infty$.

123.2 Solution 1: ϵ -Greedy

The simplest approach to balancing exploration and exploitation.

ϵ -Greedy Algorithm

At each time step:

- With probability $1 - \epsilon$: choose action with highest estimated reward (exploit)
- With probability ϵ : choose uniformly at random (explore)

Typically ϵ is small (e.g., 0.05 or 0.1).

Variant: Decaying $\epsilon_t = \min(1, c/t)$ reduces exploration over time.

ϵ -Greedy: Pros and Cons

Advantages:

- Extremely simple to implement
- Works with any reward estimator
- Robust to model misspecification

Disadvantages:

- Explores uniformly, even when some arms are clearly bad
- Regret is linear in T for fixed ϵ
- Tuning ϵ is non-trivial
- The exploration rate is constant—we pay the same exploration cost early (when exploration is valuable) and late (when we should mostly exploit)

123.3 Solution 2: Upper Confidence Bound (UCB)

A more intelligent approach that adapts exploration based on uncertainty.

UCB Algorithm

At each time step, choose action maximising:

$$\bar{r}_a + \sqrt{\frac{2 \log t}{n_a}}$$

- \bar{r}_a : Average observed reward of action a (**exploitation term**)
- $\sqrt{\frac{2 \log t}{n_a}}$: Uncertainty bonus (**exploration term**)
- n_a : Number of times action a has been played
- t : Total number of time steps so far

Principle: “Optimism in the face of uncertainty”—give underexplored actions the benefit of the doubt.

UCB Derivation Sketch

The exploration bonus comes from Hoeffding's inequality. With probability $\geq 1 - \delta$:

$$\mu_a \leq \bar{r}_a + \sqrt{\frac{\log(1/\delta)}{2n_a}}$$

Setting $\delta = 1/t^2$ and applying a union bound over arms gives the $\sqrt{2\log t/n_a}$ bonus.

This is an **upper confidence bound** on the true mean—the true value is likely below this bound.

Unpacking the UCB formula:

- **Large when n_a small:** Arms we have rarely tried get a big bonus
- **Decreases as n_a grows:** As we learn more about an arm, the bonus shrinks
- **Grows (slowly) with t :** Even well-explored arms get revisited occasionally

UCB Properties

- **Regret bound:** $O(\sqrt{KT \log T})$ where K is the number of arms
- **Early behaviour:** Explores widely (uncertainty term dominates)
- **Late behaviour:** Exploits best arm (reward term dominates)
- **Never stops exploring:** The $\log t$ term grows forever, so even well-explored arms occasionally get revisited

When UCB may struggle: UCB assumes a stationary environment where reward distributions do not change. In non-stationary settings, ϵ -greedy (with its constant exploration) may adapt better.

123.4 Solution 3: Thompson Sampling

A Bayesian alternative that achieves the same theoretical guarantees as UCB but through a different mechanism.

Thompson Sampling

Maintain a **posterior distribution** over each arm's reward.

At each step:

1. Sample from each arm's posterior: $\tilde{\mu}_a \sim p(\mu_a | \text{data})$
2. Choose the arm with highest sample: $a_t = \arg \max_a \tilde{\mu}_a$
3. Observe reward, update posterior

Key insight: Exploration is automatic. Uncertain arms have high-variance posteriors, so they sometimes produce high samples and get selected.

Thompson Sampling: Beta-Bernoulli Example

For binary rewards $r \in \{0, 1\}$:

- Prior: $\mu_a \sim \text{Beta}(\alpha_0, \beta_0)$ (e.g., $\alpha_0 = \beta_0 = 1$)
- After s_a successes and f_a failures: $\mu_a | \text{data} \sim \text{Beta}(\alpha_0 + s_a, \beta_0 + f_a)$

Algorithm:

1. Sample $\tilde{\mu}_a \sim \text{Beta}(\alpha_0 + s_a, \beta_0 + f_a)$ for each arm
2. Play $a_t = \arg \max_a \tilde{\mu}_a$
3. If reward is 1, increment s_{a_t} ; else increment f_{a_t}

Thompson Sampling Advantages

- Also achieves $O(\sqrt{KT \log T})$ regret
- Natural Bayesian interpretation
- Adapts well to non-stationary environments (posteriors can incorporate forgetting)
- Empirically often outperforms UCB
- Easy to incorporate prior knowledge
- Naturally maintains calibrated uncertainty estimates
- Often the default choice in practice (e.g., A/B testing platforms)

Thompson Sampling vs UCB

UCB: Deterministically chooses the arm with the highest upper confidence bound. The exploration is “forced” by the confidence bonus.

Thompson Sampling: Stochastically chooses arms, with exploration emerging naturally from posterior uncertainty. No explicit exploration bonus is needed.

Both achieve the same asymptotic regret bounds, but Thompson Sampling often performs better in practice and generalises more easily to complex settings.

Comparison of Bandit Algorithms

	ϵ-Greedy	UCB	Thompson
Regret	$O(T)$ or $O(\sqrt{T})$	$O(\sqrt{KT \log T})$	$O(\sqrt{KT \log T})$
Exploration	Random	Deterministic	Stochastic
Framework	Frequentist	Frequentist	Bayesian
Requires prior	No	No	Yes
Computational cost	Low	Low	Low-Medium

124 Estimating Prevalence: AIPW

Section Summary

When the goal is estimating a population quantity (not building a predictor), AIPW combines model predictions with sampling corrections. It achieves the best of both worlds: low variance from the model, unbiasedness from inverse probability weighting. The “double robustness” property means the estimator is consistent if *either* the outcome model *or* the propensity score is correctly specified.

Finally, we consider a different goal: estimating a population quantity (e.g., the fraction of online comments that are hate speech) rather than building a predictor or making decisions.

124.1 The Problem

The Prevalence Estimation Challenge

Suppose we want to estimate how common a trait is in a population, but measuring the trait is expensive.

Example: What fraction of tweets contain hate speech?

- We can easily access millions of tweets (X)
- But determining if each is hate speech (y) requires human review
- We can only afford to label a small subset

Two natural approaches, each with drawbacks:

Approach 1: Random Sampling

- Randomly sample tweets, have humans label them, compute the average
- **Pro:** Unbiased estimate of the true prevalence
- **Con:** High variance—need many labels for a precise estimate

Approach 2: Model Predictions

- Train a classifier on a labelled subset, then average predictions on the full population
- **Pro:** Low variance—can predict on millions of tweets
- **Con:** Biased if the model is miscalibrated (which it usually is)

The AIPW Solution

AIPW (Augmented Inverse Propensity Weighting) combines both approaches:

- Use model predictions for variance reduction
- Correct for model bias using labelled examples
- Achieve low variance AND unbiased estimates

124.2 Targeted Sampling for Prevalence

An important insight: trait prevalence is often not uniformly distributed. Hate speech might be concentrated in certain topics or communities. This suggests a targeted sampling strategy.

Model-Based Sampling

Rather than uniform random sampling:

1. Train a preliminary model $f(x)$ to predict the trait
2. Sample more frequently from units where $f(x)$ is high (where the trait is predicted to be more common)
3. This gives more labelled examples of the trait, reducing variance in estimates of the trait-positive subpopulation

Example scheme:

- Sample with probability p if $f(x) \leq 0.5$
- Sample with probability $2p$ if $f(x) > 0.5$

This targeted sampling must then be corrected for via inverse propensity weighting.

124.3 The AIPW Estimator

Augmented Inverse Propensity Weighting

Let:

- $\hat{g}(X)$: Model prediction of Y given X
- $\pi(X)$: Probability of being sampled (propensity score)
- R : Indicator for whether unit was sampled ($R = 1$ if labelled)
- Y : True label (observed only if $R = 1$)

The AIPW estimator for unit i :

$$\hat{Y}_{\text{AIPW},i} = \hat{g}(X_i) + \frac{R_i}{\pi(X_i)}(Y_i - \hat{g}(X_i))$$

For the population mean:

$$\hat{\mu}_{\text{AIPW}} = \frac{1}{N} \sum_{i=1}^N \left[\hat{g}(X_i) + \frac{R_i}{\pi(X_i)}(Y_i - \hat{g}(X_i)) \right]$$

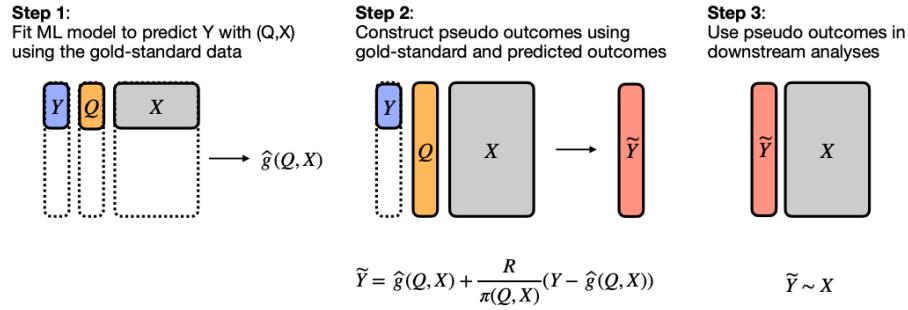


Figure 70: The AIPW workflow: (1) Fit a model on gold-standard labelled data, (2) Construct pseudo-outcomes that combine model predictions with IPW corrections, (3) Use pseudo-outcomes for population inference.

Understanding AIPW

Rewrite the AIPW estimator:

$$\hat{Y}_{\text{AIPW}} = \underbrace{\hat{g}(X)}_{\text{Model prediction}} + \underbrace{\frac{R}{\pi(X)}}_{\text{IPW}} \cdot \underbrace{(Y - \hat{g}(X))}_{\text{Prediction error}}$$

Interpretation:

- Start with the model prediction for everyone
- For labelled units, add a correction for the prediction error
- The IPW term ensures the correction is unbiased despite non-random sampling

When $R = 0$ (unit not sampled), the formula simplifies to $\hat{g}(X)$ —we use the model prediction.

When $R = 1$ (unit sampled), we get $\hat{g}(X) + \frac{1}{\pi(X)}(Y - \hat{g}(X))$ —the model prediction plus an IPW-corrected residual.

124.4 Step-by-Step AIPW Process

Step 1: Model Fitting

- Train a model $\hat{g}(X)$ to predict Y from X using a labelled “gold standard” dataset
- This model provides predictions for all units, including unlabelled ones

Step 2: Pseudo-Outcome Construction

- For each unit, compute the pseudo-outcome: $\hat{Y} = \hat{g}(X) + \frac{R}{\pi(X)}(Y - \hat{g}(X))$
- This combines the model prediction with an IPW-corrected residual
- The correction term $\frac{R}{\pi(X)}(Y - \hat{g}(X))$ is zero for unlabelled units and corrects for model errors in labelled units

Step 3: Population Inference

- Average the pseudo-outcomes across the population

- This gives an unbiased estimate of the true population mean

Why AIPW Works: Two Scenarios

If model is perfect: $Y = \hat{g}(X)$, so the correction term is zero. Use the model on everyone—no need for labels.

If model is wrong: The correction term fixes the bias using labelled data, weighted by IPW.

Double robustness: AIPW is consistent (converges to the true value) if:

- The outcome model $\hat{g}(X)$ is correctly specified, OR
- The propensity score $\pi(X)$ is correctly specified

You only need to get one of them right!

Double Robustness Explained

Why is this valuable? In practice, we rarely know the true model. Double robustness gives us two chances to get it right, making AIPW more reliable than alternatives that rely on a single model being correct.

Contrast with alternatives:

- **Pure IPW:** Requires correct propensity score; model does not help
- **Pure model-based:** Requires correct outcome model; propensity does not help
- **AIPW:** Works if either is correct

Variance Reduction

Under regularity conditions, AIPW achieves the **semiparametric efficiency bound**—it has the lowest possible variance among regular asymptotically linear estimators.

The variance reduction from using \hat{g} can be substantial:

$$\text{Var}(\hat{\mu}_{\text{AIPW}}) \approx \text{Var}(\hat{\mu}_{\text{IPW}}) \cdot (1 - R^2)$$

where R^2 is the variance explained by the model. A good model (R^2 near 1) dramatically reduces variance.

NB!**The Best of Both Worlds:**

AIPW delivers the **variance-reducing** benefits of using model predictions on every unit, combined with the **unbiasedness** guarantee of inverse propensity weighting.

This makes it the method of choice when:

- You need to estimate population quantities, not just make predictions
- Labelling is expensive but you have access to features for the whole population
- You want robustness to model misspecification

125 Summary

Key Concepts from Week 8b

Three perspectives on sampling:

1. Sampling for Better Models:

- **Active learning:** Iteratively label the most informative points
- **Uncertainty sampling:** Label where the model is uncertain
- **BALD:** Target epistemic (reducible) uncertainty, not aleatoric (irreducible) uncertainty
- **IPW correction:** Reweight non-uniform samples to recover population risk
- **Leverage scores:** Sample influential points for efficient regression ($O(p \log n / \epsilon^2)$ samples suffice)
- **Random Fourier Features:** Approximate kernels in $O(R^3)$ instead of $O(n^3)$

2. Sampling for Better Decisions (Multi-Armed Bandits):

- **Exploration-exploitation tradeoff:** Balance learning and earning
- **ϵ -greedy:** Simple but inefficient constant exploration
- **UCB:** “Optimism under uncertainty”—give underexplored arms a bonus
- **Thompson Sampling:** Bayesian approach, exploration from posterior sampling
- All achieve sublinear regret: mistakes per round $\rightarrow 0$

3. Sampling for Prevalence Estimation (AIPW):

- Combine model predictions (low variance) with IPW correction (unbiased)
- Double robustness: consistent if either model or propensity is correct
- Best of both worlds for population inference

Additional Key Concepts

1. **Data leakage:** Using information unavailable at prediction time; detect via suspiciously good performance; prevent by understanding deployment context
2. **Sampling schemes:** SRS for homogeneous populations; stratified for important subgroups; cluster for practical constraints; systematic with caution
3. **Importance sampling:** Reweight samples to estimate expectations under different distributions; beware of variance from extreme weights
4. **Heteroskedasticity:** Non-constant variance invalidates OLS standard errors; use WLS or robust standard errors

NB!**Critical Reminders:**

- **Data leakage:** Never use information at training time that will not be available at prediction time
- **Test data must be random:** Non-uniform training samples can be corrected; non-uniform test samples cannot
- **Match your method to your goal:** Model accuracy, decision quality, and prevalence estimation require different approaches

126 Chapter Overview

Section Summary: Uncertainty in ML

This chapter covers principled approaches to uncertainty quantification:

- **Why uncertainty matters:** Epistemic vs aleatoric uncertainty; calibration
- **Gaussian Processes:** Full Bayesian treatment of regression with calibrated uncertainty
- **Conformal Prediction:** Distribution-free prediction intervals with coverage guarantees
- **Bayesian Neural Networks:** Brief overview of uncertainty in deep learning
- **Calibration:** Measuring and improving reliability of uncertainty estimates

Point predictions alone are often insufficient. A medical diagnosis system that outputs “malignant” without any indication of confidence is far less useful—and potentially dangerous—compared to one that says “malignant with 95% confidence” or “uncertain, recommend further tests.” This chapter develops the mathematical machinery for principled uncertainty quantification.

In machine learning, we often focus on finding the “best” prediction—our single best guess for the output given an input. But knowing *how confident* we should be in a prediction is frequently just as important as the prediction itself. Consider a self-driving car deciding whether to brake: a prediction of “pedestrian present” with 99% confidence demands immediate action, while the same prediction with 55% confidence might warrant a more cautious response. Similarly, in medical diagnosis, a prediction of “malignant” with high confidence leads to different treatment decisions than one with substantial uncertainty.

This week introduces multiple complementary approaches to quantifying predictive uncertainty, each with distinct strengths and trade-offs. We will see that there is no single “best” method—the right choice depends on your assumptions, computational resources, and what kind of uncertainty information you need.

127 Why Uncertainty Matters

127.1 Two Fundamental Types of Uncertainty

Before diving into methods, we need to understand that not all uncertainty is created equal. There are fundamentally different *sources* of uncertainty, and distinguishing between them has practical implications for what we can do about it.

Epistemic vs Aleatoric Uncertainty

Epistemic uncertainty (model uncertainty): Uncertainty due to lack of knowledge. This uncertainty is *reducible*—it decreases as we gather more data.

Examples:

- Uncertainty about model parameters given limited training data
- Uncertainty in regions of input space with few observations
- Model misspecification (wrong functional form)

Aleatoric uncertainty (data uncertainty): Inherent randomness in the data-generating process. This uncertainty is *irreducible*—it cannot be reduced by collecting more data.

Examples:

- Measurement noise in sensors
- Inherent stochasticity in the outcome (e.g., dice rolls)
- Unobserved variables that affect the outcome

Unpacking the terminology: The word “epistemic” comes from the Greek *episteme* (knowledge)—epistemic uncertainty reflects gaps in our knowledge that could, in principle, be filled. “Aleatoric” comes from the Latin *alea* (dice)—aleatoric uncertainty is the randomness inherent in the world itself.

Why the distinction matters practically: If uncertainty is primarily epistemic, we should collect more data or improve our model. If it is aleatoric, we should communicate the inherent limits of predictability to stakeholders. Knowing which type dominates tells us whether investing in more data will help.

Mathematical formulation: In regression with $y = f(x) + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma^2)$:

- Uncertainty about $f(x)$ is epistemic (reduces with more data)
- The noise variance σ^2 is aleatoric (irreducible)

Gaussian Processes naturally separate these: the posterior variance of $f(x_*)$ captures epistemic uncertainty, while σ^2 captures aleatoric uncertainty. This separation is one of the conceptual advantages of the Bayesian framework.

A concrete example: Suppose you are predicting house prices. Epistemic uncertainty might arise because you have few examples of houses with swimming pools in your training data—if you collected more such examples, your predictions for pool-houses would become more confident. Aleatoric uncertainty arises because two identical houses in the same neighbourhood might sell for different prices due to factors you cannot observe (timing, negotiation skill, buyer preferences). No amount of additional data will eliminate this randomness.

127.2 Applications Requiring Uncertainty

When Uncertainty is Essential

1. **Safety-critical systems:** Medical diagnosis, autonomous vehicles, structural engineering—wrong predictions with high confidence are catastrophic
2. **Active learning:** Select which examples to label by targeting high-uncertainty regions
3. **Bayesian optimisation:** Balance exploration (uncertain regions) with exploitation (promising regions)
4. **Decision-making under risk:** Portfolio optimisation, resource allocation, treatment selection
5. **Outlier/anomaly detection:** Flag inputs where the model is uncertain
6. **Model debugging:** Identify where the model needs improvement

Let us expand on a few of these:

Active learning: When labelling data is expensive (e.g., requiring expert annotation), we want to be strategic about which examples to label next. A model that knows where it is uncertain can guide us to label the most informative examples. This is far more efficient than random sampling.

Bayesian optimisation: When evaluating a function is expensive (e.g., training a neural network with certain hyperparameters), we want to find the optimum with as few evaluations as possible. A model with calibrated uncertainty lets us balance trying promising regions (exploitation) against exploring uncertain regions where something better might lurk (exploration).

Anomaly detection: A model should be uncertain about inputs that are unlike anything in its training data. If a credit card fraud detector sees a transaction pattern unlike any it has seen before, high uncertainty is the appropriate response—rather than confidently (and possibly incorrectly) classifying it as normal or fraudulent.

127.3 What is Calibration?

Having uncertainty estimates is not enough—they must be *reliable*. A model that always outputs 90% confidence but is correct only 50% of the time is dangerously miscalibrated.

Calibration Definition

A probabilistic predictor is **calibrated** if its predicted probabilities match empirical frequencies.

For regression: if the model outputs a 95% prediction interval, it should contain the true value approximately 95% of the time.

For classification: among all instances where the model predicts class 1 with probability 0.8, approximately 80% should actually be class 1.

Formally, for classification with predicted probability \hat{p} :

$$P(Y = 1 \mid \hat{p} = p) = p \quad \text{for all } p \in [0, 1]$$

Unpacking the formal definition: The equation $P(Y = 1 \mid \hat{p} = p) = p$ says: “Among all the examples where I predicted probability p , the fraction that are actually positive should be p .” If

I predict 0.7 for 100 examples, approximately 70 of them should be positive.

Calibration vs accuracy: A model can be accurate but poorly calibrated. Modern neural networks, for instance, tend to be *overconfident*—they output high probabilities even when wrong. Conversely, a model could be well-calibrated but inaccurate (imagine a model that always predicts 50% for a 50-50 task—it is perfectly calibrated but useless). Calibration is about the *reliability* of uncertainty estimates, not just their existence.

Why calibration matters: If a doctor uses a diagnostic model and it says “95% probability of disease,” the doctor needs to trust that this really means 95%. If the model is overconfident and such predictions are actually correct only 60% of the time, the doctor’s decisions will be poorly informed.

128 Gaussian Processes

Section Summary: Gaussian Processes

- GPs define distributions over functions, not just parameters
- Fully specified by mean function $m(x)$ and kernel $k(x, x')$
- Key insight: any finite collection of function values is jointly Gaussian
- Posterior is available in closed form via Gaussian conditioning
- Uncertainty naturally increases away from training data

128.1 The Core Idea: Distributions Over Functions

Standard supervised learning finds a single function \hat{f} that best fits the data. The Bayesian approach instead maintains a *distribution* over all functions consistent with our prior beliefs and the observed data. This is a fundamental conceptual shift: rather than committing to one function, we maintain uncertainty about which function generated the data.

The parametric vs non-parametric distinction: In parametric models like linear regression, we have a finite number of parameters $(\beta_0, \beta_1, \dots, \beta_d)$, and the function is determined once we know these parameters. In a Gaussian Process, we place a prior directly over the space of functions. This is “non-parametric” not because there are no parameters, but because the number of effective parameters grows with the data.

Gaussian Process Definition

A **Gaussian Process** is a collection of random variables, any finite subset of which has a joint Gaussian distribution.

Equivalently: f is a GP if for any finite set of inputs x_1, \dots, x_n , the vector $(f(x_1), \dots, f(x_n))^\top$ is multivariate Gaussian.

A GP is fully specified by:

- **Mean function:** $m(x) = \mathbb{E}[f(x)]$
- **Covariance function (kernel):** $k(x, x') = \text{Cov}(f(x), f(x')) = \mathbb{E}[(f(x) - m(x))(f(x') - m(x'))]$

We write: $f \sim \mathcal{GP}(m, k)$

Unpacking the definition: The definition says that if we pick any finite collection of input points—say $x_1 = 0.5, x_2 = 1.7, x_3 = 2.3$ —then the corresponding function values ($f(0.5), f(1.7), f(2.3)$) form a multivariate Gaussian random vector. The mean of this Gaussian is given by the mean function evaluated at these points, and the covariance matrix is given by the kernel evaluated at all pairs of points.

Why this is powerful: This definition lets us work with infinite-dimensional objects (functions) using finite-dimensional Gaussian machinery. We never need to represent the entire function—only its values at the points we care about. When we want to make predictions at new points, we just extend our finite collection to include those points and use standard Gaussian conditioning.

Intuition from 3 _ reviewed: Think of a GP as a very smart, very flexible curve that we fit through data—one that does not just go straight, or curve in preset ways like polynomials. It can adopt an infinite number of shapes, guided by the properties of the data and the assumptions we encode in the kernel. Given some data points, a GP helps you predict where other points might lie, along with how certain it is about those predictions.

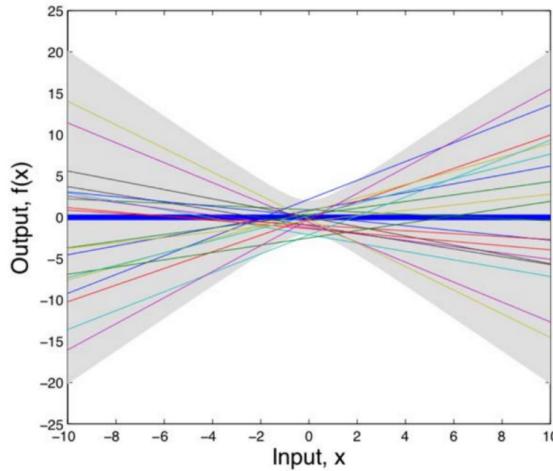


Figure 71: A GP defines a distribution over functions. The shaded region shows pointwise uncertainty (not a confidence band for a single function, but the marginal uncertainty at each point). The solid line is the posterior mean—our best estimate—while the shaded region indicates where the true function might plausibly lie.

128.2 The Bayesian Perspective: Distributions Over Functions

From Parameters to Functions

Probability distributions, commonly applied to random variables, can also be extended to **entire functions**.

Hypothesis class \mathcal{H} : the set of all functions that your algorithm can possibly learn. Instead of picking one single function as our guess, we assign a probability distribution over the entire hypothesis class—every function $f \in \mathcal{H}$ gets a probability $p(f)$ reflecting our belief in how likely it is to be the true function.

This is the Bayesian approach: we update our beliefs about which functions are likely to be good explanations for the data, rather than searching for a single “best” function.

Connection to ridge regression: Recall from Week 6 that ridge regression can be viewed as placing a Gaussian prior on the coefficients β . Coefficients near zero are considered more probable than large values, reflecting a prior belief that the true relationship is likely simple.

This prior on β induces a distribution over linear functions:

$$p(f) = p(\beta)$$

GPs generalise this idea to arbitrary (kernel-defined) function spaces. Where ridge regression places a prior over the finite-dimensional parameter space, GPs place priors directly over the infinite-dimensional space of functions. This is a natural extension of the Bayesian linear regression framework to non-linear function spaces.

128.3 Mean and Covariance Functions

Mean function $m(x)$: Encodes prior belief about the average function value at each point. In practice, often set to zero or a constant—the kernel provides enough flexibility, and centering the data removes the need for a complex mean function.

Why zero mean is common: Setting $m(x) = 0$ is not an assertion that we expect the function to be zero everywhere. Rather, it is a simplification saying that, before seeing data, we do not prefer positive values over negative values at any point. Once we observe data, the posterior mean will be non-zero where the data suggests it should be. If we have strong prior knowledge (e.g., we know the function should be roughly linear with positive slope), we can encode this in the mean function.

Covariance function $k(x, x')$: The kernel is where the real modelling power lies. It encodes our assumptions about function properties:

- **Smoothness:** How rapidly can the function change?
- **Periodicity:** Does the function repeat?
- **Amplitude:** What is the typical magnitude of function values?
- **Length-scale:** Over what distance do function values become uncorrelated?

What the kernel captures intuitively:

- How does the output value at one point relate to the output at another point?
- Do we expect smooth variations or abrupt changes?
- Are there repeating patterns?

The GP uses the kernel to measure similarity between points. Points that are “close” according to the kernel will have similar outputs. This is the same kernel concept from Week 6 (kernel ridge regression)—the kernel determines what kinds of functions are probable before seeing data.

Kernel Requirements

A function $k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ is a valid covariance function if and only if it is **positive semi-definite**: for any finite set $\{x_1, \dots, x_n\}$, the matrix K with $K_{ij} = k(x_i, x_j)$ is positive semi-definite.

Equivalently, for any n , any x_1, \dots, x_n , and any $c_1, \dots, c_n \in \mathbb{R}$:

$$\sum_{i=1}^n \sum_{j=1}^n c_i c_j k(x_i, x_j) \geq 0$$

Why positive semi-definiteness? A covariance matrix must have non-negative eigenvalues (you cannot have negative variance). The positive semi-definiteness requirement ensures that any covariance matrix we construct from the kernel will be a valid covariance matrix. This is not just a technicality—it is what makes the GP machinery work.

128.4 Sampling from a GP Prior

To understand what a GP prior “believes” about functions, we can draw samples. This is an excellent way to build intuition about what different kernels encode.

1. Choose a finite grid of inputs x_1, \dots, x_n
2. Compute the covariance matrix K with $K_{ij} = k(x_i, x_j)$
3. Sample $\mathbf{f} \sim \mathcal{N}(m(\mathbf{x}), K)$
4. Plot the sampled function values

Different kernels produce dramatically different sample functions—this is how the kernel “encodes” our prior beliefs. A smooth kernel produces smooth samples; a periodic kernel produces periodic samples. By examining prior samples, we can check whether our kernel choice matches our beliefs about the problem.

128.5 GP Prior to Posterior: The Key Insight

The magic of GPs lies in how we update from prior to posterior. The key insight is that everything remains Gaussian, so we can use standard formulas for conditioning multivariate Gaussians.

Joint Distribution of Training and Test Points

Given training inputs $X = (x_1, \dots, x_n)^\top$ with noisy observations $\mathbf{y} = (y_1, \dots, y_n)^\top$ where $y_i = f(x_i) + \epsilon_i$ and $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$, and test inputs $X_* = (x_{*1}, \dots, x_{*m})^\top$, the joint distribution is:

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} m(X) \\ m(X_*) \end{bmatrix}, \begin{bmatrix} K_{XX} + \sigma^2 I & K_{X*} \\ K_{*X} & K_{**} \end{bmatrix}\right)$$

where:

- $K_{XX} = k(X, X)$ is $n \times n$: covariance among training points
- $K_{X*} = k(X, X_*)$ is $n \times m$: covariance between training and test
- $K_{*X} = K_{X*}^\top$ is $m \times n$
- $K_{**} = k(X_*, X_*)$ is $m \times m$: covariance among test points
- $\sigma^2 I$ accounts for observation noise (only on training points)

Interpreting the block structure:

- K_{XX} : How training points relate to each other according to our kernel
- K_{X*} and K_{*X} : How much the function values at training points inform us about function values at test points—this is the “bridge” that lets us use training data to predict at new locations
- K_{**} : How much we expect function values at test points to covary with each other, *before* seeing any training data
- $\sigma^2 I$: Accounts for noise in the observed outputs—we observe $y = f(x) + \epsilon$, not $f(x)$ directly

The observation noise appears only in the top-left block because we observe $y = f(x) + \epsilon$, not $f(x)$ directly. The test points \mathbf{f}_* represent the noise-free function values.

3.2.3 Marginals and conditionals of an MVN *

Suppose $\mathbf{y} = (\mathbf{y}_1, \mathbf{y}_2)$ is jointly Gaussian with parameters

$$\boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{pmatrix}, \quad \boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{pmatrix}, \quad \boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1} = \begin{pmatrix} \boldsymbol{\Lambda}_{11} & \boldsymbol{\Lambda}_{12} \\ \boldsymbol{\Lambda}_{21} & \boldsymbol{\Lambda}_{22} \end{pmatrix}$$

where $\boldsymbol{\Lambda}$ is the **precision matrix**. Then the marginals are given by

$$\begin{aligned} p(\mathbf{y}_1) &= \mathcal{N}(\mathbf{y}_1 | \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_{11}) \\ p(\mathbf{y}_2) &= \mathcal{N}(\mathbf{y}_2 | \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_{22}) \end{aligned}$$

and the posterior conditional is given by

$$\begin{aligned} p(\mathbf{y}_1 | \mathbf{y}_2) &= \mathcal{N}(\mathbf{y}_1 | \boldsymbol{\mu}_{1|2}, \boldsymbol{\Sigma}_{1|2}) \\ \boldsymbol{\mu}_{1|2} &= \boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}(\mathbf{y}_2 - \boldsymbol{\mu}_2) \\ &= \boldsymbol{\mu}_1 - \boldsymbol{\Lambda}_{11}^{-1}\boldsymbol{\Lambda}_{12}(\mathbf{y}_2 - \boldsymbol{\mu}_2) \\ &= \boldsymbol{\Sigma}_{1|2}(\boldsymbol{\Lambda}_{11}\boldsymbol{\mu}_1 - \boldsymbol{\Lambda}_{12}(\mathbf{y}_2 - \boldsymbol{\mu}_2)) \\ \boldsymbol{\Sigma}_{1|2} &= \boldsymbol{\Sigma}_{11} - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21} = \boldsymbol{\Lambda}_{11}^{-1} \end{aligned}$$

Figure 72: The joint Gaussian over training and test points. Conditioning on training data transforms the prior into a posterior. The left shows the marginal (prior) distribution; the right shows the conditional (posterior) distribution after observing data.

128.6 Derivation of the Posterior Predictive Distribution

We now derive the posterior predictive distribution using standard Gaussian conditioning. This is the central calculation in GP regression, and understanding it deeply is essential.

Gaussian Conditioning Formula

If $\begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} \sim \mathcal{N} \left(\begin{bmatrix} \boldsymbol{\mu}_x \\ \boldsymbol{\mu}_y \end{bmatrix}, \begin{bmatrix} \boldsymbol{\Sigma}_{xx} & \boldsymbol{\Sigma}_{xy} \\ \boldsymbol{\Sigma}_{yx} & \boldsymbol{\Sigma}_{yy} \end{bmatrix} \right)$, then:

$$\mathbf{x} | \mathbf{y} \sim \mathcal{N} \left(\boldsymbol{\mu}_x + \boldsymbol{\Sigma}_{xy}\boldsymbol{\Sigma}_{yy}^{-1}(\mathbf{y} - \boldsymbol{\mu}_y), \boldsymbol{\Sigma}_{xx} - \boldsymbol{\Sigma}_{xy}\boldsymbol{\Sigma}_{yy}^{-1}\boldsymbol{\Sigma}_{yx} \right)$$

This formula is the workhorse of Gaussian inference. It tells us that when we condition a joint Gaussian on observing some of the variables, the conditional distribution is also Gaussian, with mean and covariance given by these formulas. The mean shifts based on how the observed values deviate from their prior mean, weighted by the correlation structure. The variance decreases (we become more certain) by an amount determined by how much the observed variables “explain” the unobserved ones.

Derivation of GP Posterior

Starting from the joint distribution and applying the conditioning formula with $\mathbf{x} = \mathbf{f}_*$ and $\mathbf{y} = \mathbf{y}$ (the observations):

Step 1: Identify the components:

$$\begin{aligned}\boldsymbol{\mu}_x &= m(X_*), \quad \boldsymbol{\mu}_y = m(X) \\ \Sigma_{xx} &= K_{**}, \quad \Sigma_{yy} = K_{XX} + \sigma^2 I \\ \Sigma_{xy} &= K_{*X}, \quad \Sigma_{yx} = K_{X*}\end{aligned}$$

Step 2: Apply the conditioning formula:

$$\begin{aligned}\mathbb{E}[\mathbf{f}_* | X, \mathbf{y}, X_*] &= m(X_*) + K_{*X}(K_{XX} + \sigma^2 I)^{-1}(\mathbf{y} - m(X)) \\ \text{Cov}(\mathbf{f}_* | X, \mathbf{y}, X_*) &= K_{**} - K_{*X}(K_{XX} + \sigma^2 I)^{-1}K_{X*}\end{aligned}$$

Result:

$$\mathbf{f}_* | X, \mathbf{y}, X_* \sim \mathcal{N}(\boldsymbol{\mu}_*, \Sigma_*)$$

where:

$$\boldsymbol{\mu}_* = m(X_*) + K_{*X}(K_{XX} + \sigma^2 I)^{-1}(\mathbf{y} - m(X)) \quad (21)$$

$$\Sigma_* = K_{**} - K_{*X}(K_{XX} + \sigma^2 I)^{-1}K_{X*} \quad (22)$$

Understanding the posterior mean $\boldsymbol{\mu}_*$ in detail:

The formula has a beautiful interpretation. Let us break it down term by term:

- $m(X_*)$: Start with our prior mean at the test points—this is our baseline expectation before considering the training data.
- $(\mathbf{y} - m(X))$: The “residuals” or “surprise”—how much the observed outputs deviate from what we expected under the prior. If the observations exactly matched the prior mean, this would be zero and we would not adjust our predictions at all.
- K_{*X} : Measures how similar each test point is to each training point according to our kernel. Test points similar to training points will be more influenced by the training data.
- $(K_{XX} + \sigma^2 I)^{-1}$: The inverse of the regularised covariance matrix of training points. This acts as a precision-weighted normalisation that accounts for (a) how training points relate to each other, and (b) observation noise. The $\sigma^2 I$ term prevents overfitting to noisy observations.

Understanding the posterior variance Σ_* in detail:

- K_{**} : Start with prior variance—our initial uncertainty about function values at test points before seeing any data.
- $K_{*X}(K_{XX} + \sigma^2 I)^{-1}K_{X*}$: The variance “explained” by the training data. This term measures how much our uncertainty is reduced by having observed the training points. It is always non-negative (as a quadratic form), so observing data can only reduce uncertainty, never increase it.

Interpreting the GP Posterior

Posterior mean μ_* :

- Start with prior mean $m(X_*)$
- Adjust based on: (1) how similar test points are to training points (K_{*X}), and (2) how much training observations deviate from prior ($\mathbf{y} - m(X)$)
- The matrix $(K_{XX} + \sigma^2 I)^{-1}$ acts as a “precision-weighted” combination

Posterior variance Σ_* :

- Start with prior variance K_{**}
- Subtract variance “explained” by training data
- The reduction is larger when test points are similar to training points (large K_{*X})
- Variance can only decrease from prior (observing data reduces uncertainty)

128.7 Connection to Kernel Ridge Regression

This is where the relationship to Week 6 becomes precise, and where we see the added value of the Bayesian framework.

NB!

With zero mean function, the GP posterior mean is **identical** to Kernel Ridge Regression:

$$\mu_* = K_{*X}(K_{XX} + \sigma^2 I)^{-1}\mathbf{y}$$

The regularisation parameter λ in KRR corresponds to the noise variance σ^2 in the GP.

The critical difference: GPs also compute Σ_* , providing **calibrated uncertainty estimates** rather than just point predictions.

This connection provides two perspectives on the same computation:

- **KRR view:** Regularised least squares in a kernel-induced feature space. The regularisation parameter λ is a tuning parameter we choose.
- **GP view:** Bayesian inference with a Gaussian prior over functions. The noise variance σ^2 has a natural interpretation as observation noise.

What GPs add beyond KRR:

- The diagonal elements of Σ_* give variances at each predicted point, representing the model’s confidence
- Off-diagonal elements represent covariances between predictions, indicating how uncertainties are correlated
- This additional information is valuable when making decisions under uncertainty, as it provides insight into the reliability of predictions
- The GP view suggests natural ways to estimate hyperparameters (via marginal likelihood) rather than cross-validation

128.8 Posterior of Function vs Posterior Predictive

There is an important distinction between two types of uncertainty in GPs that is often confused.

Two Types of Posterior Uncertainty

Posterior of the function f_* (credible intervals):

- Uncertainty about the *true function value* at test points
- “Where might the true function lie?”
- Variance: $\Sigma_* = K_{**} - K_{*X}(K_{XX} + \sigma^2 I)^{-1}K_{X*}$
- This is epistemic uncertainty

Posterior predictive for new observation $y_* = f_* + \epsilon_*$ (prediction intervals):

- Uncertainty about a *new noisy observation*
- “Where might future observations fall?”
- Variance: $\Sigma_* + \sigma^2 I$
- Includes both epistemic (function uncertainty) and aleatoric (noise) uncertainty
- Wider than credible intervals because observations include noise

For prediction intervals, use the posterior predictive. For understanding where the model is uncertain about the underlying function, use the posterior of f .

Concrete interpretation:

- **Credible interval:** A range of values for the function at a given point that, given the prior and observed data, are believed to contain the true function value with a certain probability. A 90% credible interval means there is a 90% probability that the true function value lies within that interval.
- **Prediction interval:** A range believed to contain future observed labels with a certain probability, accounting for observation noise. A 90% prediction interval means there is a 90% probability that a future observation will fall within this interval.

Practical implication: To shift from posterior of the function to posterior predictive, add the noise variance σ^2 to the diagonal of the covariance matrix. The prediction interval is always at least as wide as the credible interval, and approaches it in the limit of zero observation noise.

128.9 Variance Behaviour: A Key Feature of GPs

One of the most attractive properties of GPs is how their uncertainty behaves—it naturally increases in regions where we have less data.

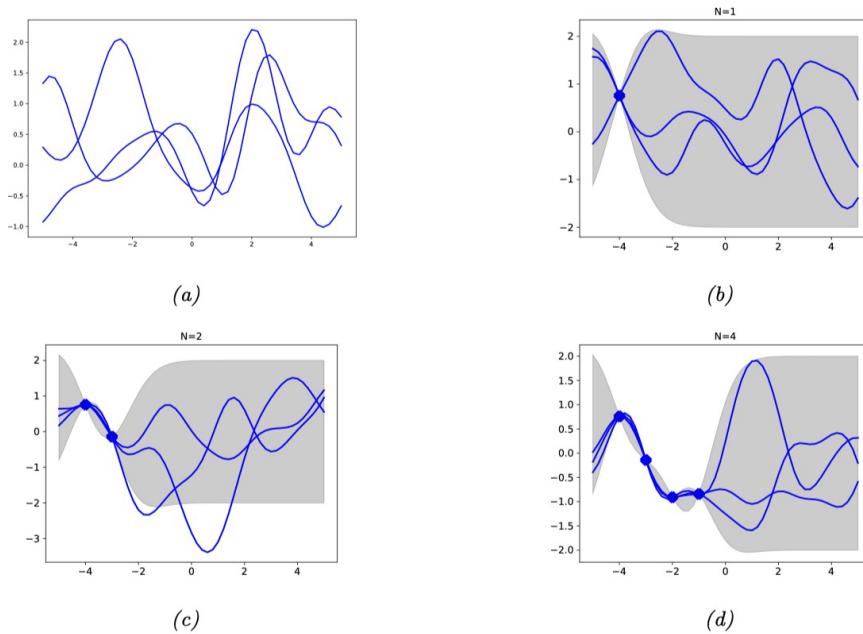


Figure 73: Evolution of GP posterior as data is observed. Functions inconsistent with observations are eliminated. The shaded region represents the confidence interval; sample functions are drawn from the posterior.

(a) No data ($n = 0$): Prior samples show high variability—we are uncertain about the function everywhere. Functions sampled from the GP prior are highly varied, and the shaded region spans a wide range reflecting that many function shapes are plausible before seeing any data.

(b) One observation ($n = 1$): Uncertainty collapses near the observed point. Sample functions are constrained to pass near the observation (within noise tolerance). The confidence interval narrows near the observed point and widens away from it. Functions inconsistent with this observation are down-weighted.

(c) Two observations ($n = 2$): Uncertainty is low near both points, higher in between and beyond. The GP “knows” more where it has seen data. Sample functions vary less near data and more in regions without observations.

(d) Four observations ($n = 4$): The posterior is well-constrained where data is dense. Uncertainty remains high only in unobserved regions. Confidence intervals are significantly tighter around all observations, and the GP has learned the trend.

GP Variance Properties

GP posterior variance has the desirable property that it:

- **Decreases near training data:** We become more certain where we have observations
- **Increases away from training data:** We remain uncertain where we lack information
- **Returns to prior variance far from all data:** In unexplored regions, we revert to prior beliefs

This behaviour is *exactly* what we want from uncertainty quantification. Many other methods (e.g., standard neural networks) do not have this property—they can be confidently wrong in extrapolation regions.

Contrast with other methods:

- Linear regression and polynomial regression provide point estimates without uncertainty measures (unless we use Bayesian versions)
- Standard neural networks often become *more* confident in extrapolation regions, which is dangerous
- This variance property makes GPs particularly valuable for active learning (sample where uncertain) and Bayesian optimisation (balance exploration and exploitation)

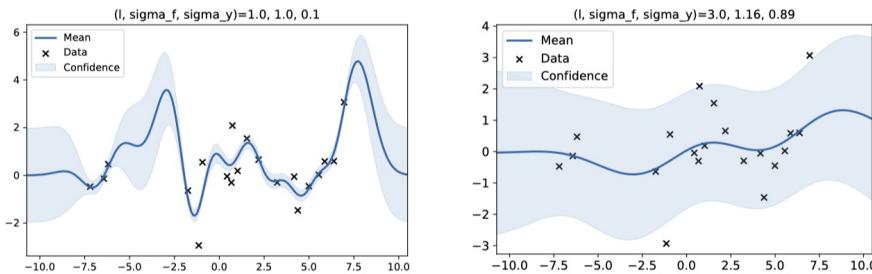


Figure 74: Effect of kernel hyperparameters on uncertainty. Crosses indicate training data. The solid line shows the mean prediction; shaded regions show confidence intervals. Larger length-scale (right) produces wider confidence bands and smoother functions, indicating that observations influence predictions over a larger range.

129 GP Kernels

Section Summary: Kernels

- The kernel encodes prior assumptions about function properties
- Common kernels: RBF (smooth), Matérn (tunable smoothness), periodic
- Kernels can be combined via addition and multiplication
- Hyperparameters are typically learned by maximising marginal likelihood

The choice of kernel is the primary way we encode prior knowledge into a GP. Different kernels encode different assumptions about function smoothness, periodicity, and other properties. Choosing the right kernel is both an art and a science—it requires understanding what each kernel implies about the functions we believe are plausible.

129.1 Squared Exponential (RBF) Kernel

The squared exponential kernel (also called the radial basis function or RBF kernel, or Gaussian kernel) is the most commonly used kernel and a good default choice.

Squared Exponential Kernel

$$k_{\text{SE}}(x, x') = \sigma_f^2 \exp\left(-\frac{\|x - x'\|^2}{2\ell^2}\right)$$

Hyperparameters:

- σ_f^2 : Signal variance (vertical scale of functions)—controls the typical magnitude of deviations from the mean
- ℓ : Length-scale (horizontal scale; how quickly correlation decays with distance)—controls how “wiggly” the functions are

Properties:

- Infinitely differentiable (very smooth functions)
- Stationary: $k(x, x') = k(x - x')$ depends only on displacement
- Universal approximator (can approximate any continuous function)

Unpacking the formula:

- $\|x - x'\|^2$: The squared Euclidean distance between inputs. Points that are far apart will have small covariance.
- $\exp(-\cdot)$: The exponential decay means covariance decreases smoothly with distance, approaching zero for very distant points.
- ℓ^2 in the denominator: Larger ℓ means the decay is slower—distant points remain correlated, producing smoother functions. Smaller ℓ means rapid decay—only nearby points are correlated, producing wigglier functions.
- σ_f^2 : This multiplier controls the overall variance. Larger σ_f^2 means functions have larger amplitude.

The length-scale ℓ controls the “wiggliness” of sampled functions:

- Small ℓ : Functions vary rapidly; nearby points can have very different values
- Large ℓ : Functions are smooth; values change slowly with input

NB!

The SE kernel produces functions that are infinitely differentiable—perhaps *too* smooth for many real-world phenomena. Physical processes often have finite smoothness. Consider Matérn kernels for more realistic smoothness assumptions.

129.2 Matérn Family

The Matérn family provides a more flexible way to control smoothness. Unlike the SE kernel, which produces infinitely smooth functions, Matérn kernels produce functions with a specified degree of differentiability.

Matérn Kernel

$$k_{\text{Matérn}}(r) = \sigma_f^2 \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu}r}{\ell} \right)^\nu K_\nu \left(\frac{\sqrt{2\nu}r}{\ell} \right)$$

where $r = \|x - x'\|$, K_ν is the modified Bessel function of the second kind, and $\nu > 0$ controls smoothness.

Special cases:

- $\nu = 1/2$: Ornstein-Uhlenbeck (continuous but not differentiable—rough, jagged functions)

$$k_{1/2}(r) = \sigma_f^2 \exp \left(-\frac{r}{\ell} \right)$$

- $\nu = 3/2$: Once differentiable (smooth but can have kinks)

$$k_{3/2}(r) = \sigma_f^2 \left(1 + \frac{\sqrt{3}r}{\ell} \right) \exp \left(-\frac{\sqrt{3}r}{\ell} \right)$$

- $\nu = 5/2$: Twice differentiable (smoother, no kinks)

$$k_{5/2}(r) = \sigma_f^2 \left(1 + \frac{\sqrt{5}r}{\ell} + \frac{5r^2}{3\ell^2} \right) \exp \left(-\frac{\sqrt{5}r}{\ell} \right)$$

- $\nu \rightarrow \infty$: Squared exponential (infinitely differentiable)

When to use Matérn: The Matérn-5/2 kernel is a popular default choice: smooth enough to be well-behaved, but not unrealistically smooth like the SE kernel. Many physical processes are well-modelled by Matérn-3/2 or Matérn-5/2 because they allow for some roughness while remaining mathematically tractable.

Interpreting ν : Functions sampled from a Matérn GP are $\lceil \nu \rceil - 1$ times mean-square differentiable. So $\nu = 3/2$ gives once-differentiable functions, $\nu = 5/2$ gives twice-differentiable functions, and so on.

129.3 Periodic Kernel

When we expect the function to repeat with a known period, the periodic kernel is appropriate.

Periodic Kernel

$$k_{\text{Per}}(x, x') = \sigma_f^2 \exp\left(-\frac{2 \sin^2(\pi|x - x'|/p)}{\ell^2}\right)$$

Hyperparameters:

- p : Period of the function
- ℓ : Length-scale (smoothness within each period)
- σ_f^2 : Signal variance

Functions sampled from this kernel are exactly periodic with period p .

When to use: Time series with known seasonal patterns (daily, weekly, yearly cycles), signals with known periodicity.

129.4 Linear Kernel

The linear kernel reduces the GP to Bayesian linear regression.

Linear Kernel

$$k_{\text{Lin}}(x, x') = \sigma_b^2 + \sigma_v^2(x - c)(x' - c)$$

where c is the offset, σ_b^2 is the bias variance, and σ_v^2 controls the variance of the slope.

A GP with a linear kernel is equivalent to Bayesian linear regression.

When to use: When you believe the relationship is linear but want uncertainty quantification. This provides a nice bridge between simple linear models and the full GP framework.

129.5 Kernel Composition

One of the beautiful aspects of kernels is that they can be combined to create more expressive kernels. This lets us build complex prior beliefs from simple building blocks.

Valid kernels can be combined to create new valid kernels:

Kernel Algebra

If k_1 and k_2 are valid kernels, so are:

- **Sum:** $k(x, x') = k_1(x, x') + k_2(x, x')$
 - Models functions that are sums of independent components
 - Example: $k_{\text{SE}} + k_{\text{Per}}$ for trend plus seasonality
- **Product:** $k(x, x') = k_1(x, x') \cdot k_2(x, x')$
 - Models functions where properties interact
 - Example: $k_{\text{SE}} \cdot k_{\text{Per}}$ for periodic with locally-varying amplitude
- **Scalar multiplication:** $k(x, x') = c \cdot k_1(x, x')$ for $c > 0$

Example: Time series with trend, seasonality, and noise:

$$k(t, t') = k_{\text{Lin}}(t, t') + k_{\text{Per}}(t, t') + k_{\text{SE}}(t, t') + \sigma^2 \delta_{tt'}$$

This decomposes the signal into interpretable components: a linear trend, a periodic seasonal component, smooth local variations, and observation noise. Each component can be analysed separately, providing interpretable structure.

129.6 Automatic Relevance Determination (ARD)

For multi-dimensional inputs, use different length-scales per dimension. This allows the model to automatically determine which input dimensions are relevant.

ARD Kernel

$$k_{\text{ARD}}(\mathbf{x}, \mathbf{x}') = \sigma_f^2 \exp \left(-\frac{1}{2} \sum_{d=1}^D \frac{(x_d - x'_d)^2}{\ell_d^2} \right)$$

Each dimension d has its own length-scale ℓ_d . During hyperparameter optimisation:

- Irrelevant dimensions: $\ell_d \rightarrow \infty$ (dimension ignored, as even large differences in x_d do not affect covariance)
- Relevant dimensions: ℓ_d stays finite

This provides automatic feature selection.

Interpretation: If a dimension is irrelevant for predicting the output, the optimised length-scale for that dimension will be very large. This effectively removes that dimension from consideration. The learned length-scales thus provide insight into which features matter.

129.7 Hyperparameter Learning

Kernel hyperparameters $\boldsymbol{\theta} = (\sigma_f, \ell, \sigma, \dots)$ are typically learned by maximising the marginal likelihood. This is a principled Bayesian approach that automatically balances fit against complexity.

Marginal Likelihood

The log marginal likelihood is:

$$\log p(\mathbf{y} | X, \boldsymbol{\theta}) = -\frac{1}{2}\mathbf{y}^\top K_y^{-1}\mathbf{y} - \frac{1}{2}\log|K_y| - \frac{n}{2}\log(2\pi)$$

where $K_y = K_{XX} + \sigma^2 I$.

Three terms:

1. $-\frac{1}{2}\mathbf{y}^\top K_y^{-1}\mathbf{y}$: Data fit (prefers parameters that explain the data)
2. $-\frac{1}{2}\log|K_y|$: Complexity penalty (prefers simpler models—this term increases with the “flexibility” of the GP)
3. $-\frac{n}{2}\log(2\pi)$: Normalisation constant (does not depend on $\boldsymbol{\theta}$)

This automatically balances fit against complexity—no separate validation set needed for hyperparameter selection.

Unpacking the tradeoff:

- The first term wants the GP to fit the data well. Very flexible GPs (small length-scale) can fit any data well.
- The second term penalises complexity. Very flexible GPs have large determinants $|K_y|$, so this term penalises overly wiggly fits.
- The balance between these terms implements Occam’s razor: among models that fit the data equally well, prefer simpler ones.

Optimisation is typically done via gradient descent. The gradients have closed forms:

$$\frac{\partial}{\partial \theta_j} \log p(\mathbf{y} | X, \boldsymbol{\theta}) = \frac{1}{2}\mathbf{y}^\top K_y^{-1} \frac{\partial K_y}{\partial \theta_j} K_y^{-1} \mathbf{y} - \frac{1}{2} \text{tr} \left(K_y^{-1} \frac{\partial K_y}{\partial \theta_j} \right)$$

NB!

The marginal likelihood can have multiple local optima. In practice:

- Use multiple random restarts
- Consider placing priors on hyperparameters (fully Bayesian treatment)
- Be aware that very short length-scales can lead to overfitting
- Very long length-scales can lead to underfitting (function is too smooth)

130 Computational Aspects of GPs

Section Summary: Computation

- Exact GP inference costs $O(n^3)$ time and $O(n^2)$ memory
- Bottleneck: inverting the $n \times n$ covariance matrix
- Sparse/inducing point methods reduce to $O(nm^2)$ where $m \ll n$
- GPs are practical for $n \lesssim 10,000$; sparse methods extend to $n \sim 10^6$

130.1 Exact GP Complexity

The computational cost of GPs is their main practical limitation.

Computational Cost

For n training points and n_* test points:

Training (computing $(K_{XX} + \sigma^2 I)^{-1}$):

- Time: $O(n^3)$ for matrix inversion (or Cholesky decomposition)
- Memory: $O(n^2)$ to store the covariance matrix

Prediction:

- Mean: $O(n)$ per test point (matrix-vector multiplication)
- Variance: $O(n^2)$ per test point (if computing full covariance)

Hyperparameter optimisation: Each gradient evaluation requires $O(n^3)$, and we need many iterations.

Why $O(n^3)$? The dominant cost is computing the Cholesky decomposition of the $n \times n$ covariance matrix, which is $O(n^3)$. We also need to store the full $n \times n$ matrix, which requires $O(n^2)$ memory.

The cubic scaling makes exact GPs impractical for large datasets:

n	Approximate time for n^3 operations
1,000	10^9 (seconds)
10,000	10^{12} (hours)
100,000	10^{15} (years)

These are rough estimates assuming one FLOP per nanosecond, but they illustrate the severity of the scaling problem.

130.2 Sparse Gaussian Processes

The key idea: approximate the full GP using a smaller set of $m \ll n$ **inducing points**. This trades some accuracy for dramatically better scaling.

Inducing Point Methods

Instead of conditioning on all n training points, we condition on m **inducing points** $Z = (z_1, \dots, z_m)^\top$ with corresponding function values $\mathbf{u} = (f(z_1), \dots, f(z_m))^\top$.

Key assumption: Given the inducing points, training and test points are conditionally independent:

$$p(f_*, \mathbf{f} \mid \mathbf{u}) \approx p(f_* \mid \mathbf{u})p(\mathbf{f} \mid \mathbf{u})$$

Cost reduction:

- Training: $O(nm^2)$ instead of $O(n^3)$
- Memory: $O(nm)$ instead of $O(n^2)$
- Prediction: $O(m^2)$ per test point

Intuition: The inducing points act as a “summary” of the training data. Instead of remembering all n training points, we summarise them with m representative points. If m is much smaller than n but large enough to capture the important structure, we get a good approximation at much lower cost.

Choosing inducing points: The locations Z can be:

- Fixed (e.g., on a grid, or a subset of training points)
- Learned jointly with hyperparameters via variational inference

Variational Sparse GPs (SVGP)

Treat the inducing point values \mathbf{u} as variational parameters. Maximise a lower bound on the marginal likelihood:

$$\log p(\mathbf{y}) \geq \mathbb{E}_{q(\mathbf{u})}[\log p(\mathbf{y} \mid \mathbf{u})] - \text{KL}(q(\mathbf{u}) \| p(\mathbf{u}))$$

where $q(\mathbf{u}) = \mathcal{N}(\mathbf{m}, \mathbf{S})$ is a variational Gaussian.

This enables:

- Stochastic optimisation (mini-batches of data)
- Scaling to millions of data points
- Learning inducing point locations

The variational approach: Instead of computing the exact posterior, we find the best Gaussian approximation within a tractable family. The ELBO (evidence lower bound) balances fitting the data against staying close to the prior. Mini-batch optimisation makes this scalable to very large datasets.

130.3 When to Use GPs

GP Applicability Guidelines

GPs are well-suited when:

- Dataset is small to medium ($n \lesssim 10,000$ for exact; larger with sparse methods)
- Calibrated uncertainty is important
- Function is expected to be smooth (or smoothness is characterisable)
- Interpretable priors are valuable
- Bayesian optimisation of expensive functions

GPs may not be ideal when:

- Dataset is very large ($n > 10^6$) and uncertainty is not critical
- Input dimension is very high ($D > 100$)—curse of dimensionality affects kernel-based methods
- Function has complex structure better captured by deep networks

131 Bayesian Optimisation

GPs are the workhorse of Bayesian optimisation—the sequential design of experiments to find the optimum of expensive-to-evaluate functions. This is one of the most successful practical applications of GPs.

Bayesian Optimisation Framework

Goal: Find $x^* = \arg \min_x f(x)$ where f is expensive to evaluate.

Algorithm:

1. Fit a GP surrogate model to observed data $\{(x_i, y_i)\}_{i=1}^n$
2. Use an **acquisition function** $\alpha(x)$ to select the next point
3. Evaluate f at the selected point; add to dataset
4. Repeat until budget exhausted

Acquisition functions balance exploration (sample where uncertain) and exploitation (sample where expected to be good):

- **Expected Improvement (EI):** $E[\max(f_{\min} - f(x), 0)]$ —the expected amount by which we will improve over the current best
- **Probability of Improvement (PI):** $P(f(x) < f_{\min})$ —the probability that we will improve (ignores magnitude of improvement)
- **Upper Confidence Bound (UCB):** $\mu(x) - \kappa\sigma(x)$ (for minimisation)—mean minus a multiple of standard deviation

Why the GP's uncertainty is crucial: The acquisition function needs to balance exploration and exploitation. The GP's uncertainty estimate tells us where we are uncertain (should explore) versus where we are confident (can exploit). Without calibrated uncertainty, the optimisation would either over-explore (wasting evaluations in already-explored regions) or over-exploit (getting stuck in local optima and missing the global optimum).

Applications:

- Hyperparameter tuning for machine learning models
- Experimental design in chemistry, materials science, drug discovery
- Engineering optimisation where simulations are expensive
- Any setting where function evaluations are costly and we want to find the optimum with minimal evaluations

Interactive demo: <http://www.infinitecuriosity.org/vizgp/>

132 Conformal Prediction

Section Summary: Conformal Prediction

- Distribution-free method for constructing prediction sets
- Provides finite-sample coverage guarantees under minimal assumptions
- Works with *any* predictive model (model-agnostic)
- Key assumption: exchangeability (weaker than i.i.d.)
- Guarantees marginal coverage, not conditional coverage

Conformal prediction offers a fundamentally different approach to uncertainty: instead of modelling the data-generating process (as GPs do), it uses the data directly to construct prediction sets with guaranteed coverage. The approach is remarkably general—it can “wrap” any predictive model and turn its point predictions into valid prediction intervals.

132.1 The Coverage Guarantee

Conformal Prediction Guarantee

Given exchangeable data $(X_1, Y_1), \dots, (X_n, Y_n), (X_{n+1}, Y_{n+1})$, conformal prediction constructs a prediction set $\mathcal{C}(X_{n+1})$ such that:

$$P(Y_{n+1} \in \mathcal{C}(X_{n+1})) \geq 1 - \alpha$$

This is a **finite-sample** guarantee—it holds for any n , not just asymptotically.

Exchangeability: A sequence is exchangeable if its joint distribution is invariant to permutations. This is weaker than i.i.d. (i.i.d. implies exchangeable, but not vice versa).

The remarkable fact: This guarantee holds regardless of the underlying distribution, the model used, or the dimensionality of the problem. No Gaussian assumptions, no asymptotic arguments. If you say “95% coverage,” you get 95% coverage.

Exchangeability explained: A sequence (Z_1, Z_2, \dots, Z_n) is exchangeable if swapping any elements does not change the joint distribution. For example, (Z_1, Z_2, Z_3) has the same distribution as (Z_2, Z_1, Z_3) or (Z_3, Z_1, Z_2) . The key insight is that i.i.d. samples are exchangeable (since each is drawn from the same distribution independently), but exchangeability is a weaker condition that accommodates some dependencies.

What is a prediction set? For regression, it is typically an interval $[l, u]$. For classification, it is a subset of class labels. The guarantee says the true value will be in this set with probability at least $1 - \alpha$.

132.2 Split Conformal Prediction

The simplest and most practical variant is **split conformal prediction**. It trades some statistical efficiency (using part of the data for calibration rather than training) for computational simplicity and clean theory.

Split Conformal Algorithm

Input: Data $\{(x_i, y_i)\}_{i=1}^n$, model class, coverage level $1 - \alpha$

Step 1: Split data

- Training set: $\mathcal{D}_{\text{train}}$ (fit the model)
- Calibration set: $\mathcal{D}_{\text{cal}} = \{(x_i, y_i)\}_{i=1}^{n_{\text{cal}}}$ (construct intervals)

Step 2: Fit model on $\mathcal{D}_{\text{train}}$ to get \hat{f}

Step 3: Define nonconformity score $s(x, y)$ measuring how “unusual” y is for input x

- Regression: $s(x, y) = |y - \hat{f}(x)|$ (absolute residual)
- Classification: $s(x, y) = 1 - \hat{f}(x)_y$ (one minus predicted probability of true class)

Step 4: Compute calibration scores

$$s_i = s(x_i, y_i) \quad \text{for } (x_i, y_i) \in \mathcal{D}_{\text{cal}}$$

Step 5: Find quantile

$$\hat{q} = \text{Quantile}_{(1-\alpha)(1+1/n_{\text{cal}})}(s_1, \dots, s_{n_{\text{cal}}})$$

(the $\lceil (n_{\text{cal}} + 1)(1 - \alpha) \rceil$ -th smallest value)

Step 6: Construct prediction set

$$\mathcal{C}(x_{\text{new}}) = \{y : s(x_{\text{new}}, y) \leq \hat{q}\}$$

For regression with absolute residual scores, this simplifies beautifully:

$$\mathcal{C}(x_{\text{new}}) = [\hat{f}(x_{\text{new}}) - \hat{q}, \hat{f}(x_{\text{new}}) + \hat{q}]$$

The prediction interval is simply the point prediction plus or minus the calibrated threshold.

Step-by-step intuition:

1. We fit our favourite model on part of the data.
2. On held-out calibration data, we see how badly the model’s predictions can be wrong (the distribution of residuals).
3. We find a threshold such that most $((1 - \alpha)$ fraction) of residuals are smaller than this threshold.
4. For new predictions, we use this threshold to construct intervals that will contain the true value with the desired probability.

132.3 Why Does It Work?

Proof Sketch of Coverage Guarantee

Under exchangeability, the calibration scores $s_1, \dots, s_{n_{\text{cal}}}$ and the new score $s_{n_{\text{cal}}+1} = s(x_{\text{new}}, y_{\text{new}})$ are exchangeable.

By symmetry, the rank of $s_{n_{\text{cal}}+1}$ among all $n_{\text{cal}} + 1$ scores is uniformly distributed over $\{1, \dots, n_{\text{cal}} + 1\}$.

Therefore:

$$P(s_{n_{\text{cal}}+1} \leq \hat{q}) = P(\text{rank}(s_{n_{\text{cal}}+1}) \leq \lceil (n_{\text{cal}} + 1)(1 - \alpha) \rceil) \geq 1 - \alpha$$

Since $y_{\text{new}} \in \mathcal{C}(x_{\text{new}})$ iff $s(x_{\text{new}}, y_{\text{new}}) \leq \hat{q}$, we have coverage.

The elegance of this proof: It relies purely on the symmetry of exchangeable random variables, not on any distributional assumptions. If we have $n + 1$ exchangeable scores, the new score is equally likely to be in any rank position from 1 to $n + 1$. The probability that it is among the smallest $\lceil (n + 1)(1 - \alpha) \rceil$ is at least $1 - \alpha$ by construction.

Why the $(1 + 1/n_{\text{cal}})$ factor? This finite-sample correction accounts for the fact that we are using a finite calibration set. As $n_{\text{cal}} \rightarrow \infty$, this correction becomes negligible.

132.4 Choice of Nonconformity Score

The score function $s(x, y)$ determines the shape and efficiency of prediction sets. The coverage guarantee holds for *any* score function, but the choice affects how tight the intervals are.

Common Score Functions

Regression:

- Absolute residual: $s(x, y) = |y - \hat{f}(x)|$
 - Simple, gives symmetric intervals
 - Width is constant across all x
- Normalised residual: $s(x, y) = \frac{|y - \hat{f}(x)|}{\hat{\sigma}(x)}$
 - Requires a model $\hat{\sigma}(x)$ of local variance
 - Gives adaptive interval widths (tighter where variance is low)
- Quantile-based: use quantile regression to estimate conditional quantiles

Classification:

- $s(x, y) = 1 - \hat{p}_y(x)$: one minus predicted probability of class y
- Prediction set includes all classes with $\hat{p}_y(x) \geq 1 - \hat{q}$

The key insight: The coverage guarantee is *valid* regardless of the score function, but smart score functions produce *tighter* intervals. The score function is where you can incorporate domain knowledge and model-specific information.

132.5 Handling Heteroskedasticity

When variance varies with x (heteroskedasticity), the absolute residual score produces intervals of constant width, which is inefficient. The normalised score produces **adaptive prediction intervals**.

1. Train a model $\hat{f}(x)$ for the conditional mean
2. Train a model $\hat{\sigma}(x)$ for the conditional standard deviation (e.g., by fitting to absolute residuals from step 1)
3. Use normalised scores: $s(x, y) = |y - \hat{f}(x)|/\hat{\sigma}(x)$
4. Find quantile \hat{q} of normalised calibration scores
5. Prediction interval: $\hat{f}(x_{\text{new}}) \pm \hat{q} \cdot \hat{\sigma}(x_{\text{new}})$

The interval width adapts to the local variance while maintaining the coverage guarantee. Where $\hat{\sigma}(x)$ is small, intervals are tight; where it is large, intervals are wide. This is exactly what we want: tighter intervals where we can be more precise, wider intervals where there is more inherent variability.

132.6 Marginal vs Conditional Coverage

NB!

Conformal prediction guarantees **marginal coverage**:

$$P(Y_{\text{new}} \in \mathcal{C}(X_{\text{new}})) \geq 1 - \alpha$$

averaged over the joint distribution of $(X_{\text{new}}, Y_{\text{new}})$.

It does *not* guarantee **conditional coverage**:

$$P(Y_{\text{new}} \in \mathcal{C}(X_{\text{new}}) \mid X_{\text{new}} = x) \geq 1 - \alpha \quad \text{for all } x$$

This is a fundamental limitation. Without distributional assumptions, conditional coverage is generally impossible to achieve.

Example of the distinction: The 90% marginal coverage guarantee can be achieved by over-covering in some regions and under-covering in others:

Region of X	Proportion of data	Coverage in region	Contribution to marginal
$X \in A$	60%	98%	58.8%
$X \in B$	40%	78%	31.2%
Marginal coverage:			90%

For a point in region B , the nominal 90% guarantee does not hold—actual coverage is only 78%. The marginal guarantee is satisfied, but conditional coverage in region B is violated.

Why this matters: If your application requires coverage guarantees for specific subgroups or regions of the input space, marginal coverage may not be sufficient. A patient from a minority group might experience lower coverage than the nominal rate.

Approaches for better conditional coverage (active research area):

- Locally-weighted conformal prediction
- Conformalized quantile regression
- Mondrian conformal prediction (coverage within predefined groups)

132.7 Example: Non-Normal Errors

Consider a data generating process $y = \beta X + \epsilon$, where ϵ has a highly non-normal distribution (e.g., heavy tails, skewness). Standard regression assumes normal errors, so classical confidence intervals would be invalid.

Even though standard regression assumes normal errors, we can “conformalise” our predictions to get a valid prediction interval:

1. **Fit your model:** Fit a linear model $\hat{y} = \hat{\beta}X$ to the training data
2. **Compute residuals:** Calculate $|y - \hat{y}|$ for each point in the calibration set
3. **Find quantile:** Determine the 95th percentile of these residuals
4. **Form interval:** Prediction interval is $\hat{y} \pm$ (95th percentile)

The interval is **valid** (95% coverage) regardless of ϵ ’s distribution. It may not be *tight* (optimal width), but it is valid. This is the power of distribution-free methods.

132.8 Comparison: Conformal vs Bayesian Approaches

Conformal vs Gaussian Processes		
	Gaussian Processes	Conformal Prediction
Assumptions	Gaussian, kernel choice	Exchangeability only
Output	Full posterior distribution	Prediction sets/intervals
Calibration	Built-in (if model correct)	Requires calibration data
Computational cost	$O(n^3)$	$O(n \log n)$ (sorting)
Model-agnostic	No	Yes
Coverage type	Conditional (if well-specified)	Marginal
Uncertainty type	Epistemic + aleatoric	Combined (implicit)

Neither approach dominates. GPs provide richer information (full distributions, separation of uncertainty types) but require stronger assumptions and are computationally expensive. Conformal prediction is more robust (works with any model, any distribution) but provides less detailed information (just intervals, not distributions) and only guarantees marginal coverage.

When to use GPs:

- Smaller datasets where modelling detailed uncertainties is crucial
- When you need the full posterior distribution, not just intervals
- For Bayesian optimisation and active learning
- When kernel choice can encode meaningful prior knowledge

When to use conformal inference:

- Large datasets where GP computation is prohibitive

- When you want model-agnostic prediction intervals
- When distribution-free guarantees are important
- As a “wrapper” around any existing predictive model

133 Bayesian Neural Networks

Section Summary: Bayesian Neural Networks

- Place distributions over neural network weights instead of point estimates
- Theoretically principled but computationally challenging
- Exact posterior is intractable; approximations are necessary
- MC Dropout: simple approximation using dropout at test time
- Deep ensembles: train multiple networks, use disagreement as uncertainty

Standard neural networks produce point predictions without principled uncertainty estimates. Bayesian neural networks (BNNs) address this by maintaining distributions over weights. However, the intractability of the posterior makes this challenging in practice.

133.1 The BNN Framework

Bayesian Neural Network

Prior: Place a prior $p(\mathbf{w})$ over network weights \mathbf{w}

Likelihood: $p(\mathcal{D} \mid \mathbf{w}) = \prod_{i=1}^n p(y_i \mid x_i, \mathbf{w})$

Posterior: Apply Bayes’ rule:

$$p(\mathbf{w} \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \mathbf{w})p(\mathbf{w})}{p(\mathcal{D})}$$

Prediction: Marginalise over the posterior:

$$p(y_* \mid x_*, \mathcal{D}) = \int p(y_* \mid x_*, \mathbf{w})p(\mathbf{w} \mid \mathcal{D}) d\mathbf{w}$$

The predictive distribution captures both epistemic uncertainty (posterior width—uncertainty about which weights are correct) and aleatoric uncertainty (inherent noise in the output given a fixed function).

Conceptual comparison with GPs: Both GPs and BNNs place distributions over functions. In GPs, we work directly with function values. In BNNs, we work with distributions over weights, which induce distributions over functions. The GP approach is analytically tractable for the Gaussian case; the BNN approach requires approximations but can leverage the representational power of neural network architectures.

133.2 The Challenge: Intractable Posterior

NB!

The posterior $p(\mathbf{w} | \mathcal{D})$ is **intractable** for neural networks:

- The normalising constant $p(\mathcal{D}) = \int p(\mathcal{D} | \mathbf{w})p(\mathbf{w}) d\mathbf{w}$ cannot be computed (the integral is over millions of dimensions)
- The posterior is highly non-Gaussian and multimodal
- Modern networks have millions of parameters

All practical BNN methods use **approximations** to the true posterior.

Why is this hard? The posterior over neural network weights lives in a space of millions of dimensions. The posterior landscape is highly complex, with many local modes corresponding to different equivalent networks (due to symmetries in the weight space). Computing the normalising constant would require integrating over all possible weight configurations—**intractable** by any method.

133.3 Approximation Methods

Variational Inference: Approximate $p(\mathbf{w} | \mathcal{D})$ with a tractable family $q_\phi(\mathbf{w})$ (e.g., factorised Gaussian). Optimise ϕ to minimise $\text{KL}(q_\phi \| p(\cdot | \mathcal{D}))$.

MC Dropout: A simple and widely-used approximation that requires minimal changes to standard neural network training.

MC Dropout

Insight (Gal & Ghahramani, 2016): Dropout applied at test time approximates Bayesian inference.

Algorithm:

1. Train network with dropout as usual
2. At test time, keep dropout enabled (normally it is disabled)
3. Run T forward passes with different dropout masks
4. Predictive mean: $\hat{\mu}(x) = \frac{1}{T} \sum_{t=1}^T f_{\mathbf{w}_t}(x)$
5. Predictive variance: $\hat{\sigma}^2(x) = \frac{1}{T} \sum_{t=1}^T (f_{\mathbf{w}_t}(x) - \hat{\mu}(x))^2$

The variance across forward passes estimates epistemic uncertainty.

Intuition: Each dropout mask corresponds to a different “thinned” network. By running multiple forward passes with different masks, we are effectively sampling from an implicit distribution over networks. The variability in predictions reflects uncertainty about which network is correct.

Advantages: No change to training; easy to implement; works with existing networks.

Limitations: The approximation quality depends on dropout rate and architecture; may underestimate uncertainty.

Deep Ensembles: Train M networks with different random initialisations. Use the ensemble mean for prediction and ensemble variance for uncertainty.

Deep Ensembles

Algorithm:

1. Train M networks independently (different random seeds)
2. Each network m outputs mean $\mu_m(x)$ and variance $\sigma_m^2(x)$
3. Ensemble prediction:

$$\mu_*(x) = \frac{1}{M} \sum_{m=1}^M \mu_m(x)$$

$$\sigma_*^2(x) = \underbrace{\frac{1}{M} \sum_{m=1}^M \sigma_m^2(x)}_{\text{aleatoric}} + \underbrace{\frac{1}{M} \sum_{m=1}^M (\mu_m(x) - \mu_*(x))^2}_{\text{epistemic}}$$

Disagreement among ensemble members indicates epistemic uncertainty.

Why this works: Different random initialisations lead networks to converge to different local minima. The variability in their predictions reflects genuine uncertainty about which solution is “correct.” Regions where all networks agree are likely well-determined by the data; regions where they disagree indicate uncertainty.

Decomposition of uncertainty: The formula elegantly separates:

- **Aleatoric uncertainty:** The average of individual network variances. This is the “inherent noise” that each network predicts, which cannot be reduced by training more networks.
- **Epistemic uncertainty:** The variance of the network means. This measures disagreement between networks, which would decrease if we had more training data.

Deep ensembles often outperform other BNN approximations in practice, despite being theoretically less principled than variational methods.

133.4 Connection to Gaussian Processes

There is a beautiful theoretical connection between neural networks and GPs.

Neural Network-GP Correspondence

Theorem (Neal, 1996; Lee et al., 2018): A single-hidden-layer neural network with:

- Random weights drawn from $\mathcal{N}(0, \sigma_w^2/n_{\text{hidden}})$
- Random biases drawn from $\mathcal{N}(0, \sigma_b^2)$
- Fixed nonlinearity ϕ

converges to a Gaussian process as the hidden layer width $n_{\text{hidden}} \rightarrow \infty$.

The kernel of the limiting GP is determined by the nonlinearity:

$$k(x, x') = \sigma_b^2 + \sigma_w^2 \mathbb{E}_{z \sim \mathcal{N}(0, \Sigma)} [\phi(z_1)\phi(z_2)]$$

where Σ is determined by x and x' .

Implications of this connection:

- Provides theoretical grounding for neural network priors
- Suggests that very wide networks may be well-approximated by GPs
- Has led to the “Neural Tangent Kernel” theory for understanding training dynamics
- Gives insight into what implicit prior assumptions neural networks make

Practical relevance: While real networks are finite-width, this theory helps us understand the function spaces that neural networks naturally inhabit and provides tools for analysing their behaviour.

134 Calibration

Section Summary: Calibration

- Calibration measures whether predicted probabilities match actual frequencies
- Reliability diagrams visualise calibration
- Expected Calibration Error (ECE) quantifies miscalibration
- Post-hoc methods (temperature scaling, Platt scaling) can improve calibration
- Modern neural networks are typically overconfident

Having uncertainty estimates is not enough—they must be *reliable*. A model that always outputs 90% confidence but is correct only 50% of the time is dangerously miscalibrated. This section covers how to measure and improve calibration.

134.1 Reliability Diagrams

Reliability diagrams are the primary visual tool for assessing calibration.

Reliability Diagram Construction

For a classifier with predicted probabilities:

1. Bin predictions by confidence level (e.g., 10 bins: $[0, 0.1)$, $[0.1, 0.2)$, \dots)
2. For each bin, compute:
 - Average predicted confidence
 - Actual accuracy (fraction of correct predictions)
3. Plot average confidence vs actual accuracy
4. Perfect calibration: points lie on the diagonal

Interpretation:

- Points above diagonal: model is **underconfident** (predictions are better than claimed)
- Points below diagonal: model is **overconfident** (predictions are worse than claimed)

Modern deep networks typically show overconfidence—points below the diagonal, especially at high confidence levels. A model might predict “99% confident” but actually be correct only 80% of the time in such cases.

134.2 Expected Calibration Error

Expected Calibration Error (ECE)

$$\text{ECE} = \sum_{b=1}^B \frac{|B_b|}{n} |\text{acc}(B_b) - \text{conf}(B_b)|$$

where:

- B_b is the set of samples in bin b
- $\text{acc}(B_b)$ is the accuracy in bin b
- $\text{conf}(B_b)$ is the average confidence in bin b
- $|B_b|/n$ weights by bin size

ECE is the weighted average gap between confidence and accuracy.

Lower ECE indicates better calibration. $\text{ECE} = 0$ means perfect calibration.

Interpretation: ECE measures, on average, how far the predicted confidence is from the actual accuracy. If a model predicts 80% confidence for a bin of examples and 85% of them are correct, that bin contributes $|85\% - 80\%| = 5\%$ to the ECE, weighted by the proportion of examples in that bin.

NB!

ECE has limitations:

- Sensitive to binning scheme (number of bins, equal-width vs equal-mass)
- Does not penalise all forms of miscalibration equally
- Can be low even with poor calibration in certain regions

Use reliability diagrams alongside ECE for a complete picture.

134.3 Temperature Scaling

The simplest and often most effective post-hoc calibration method. It requires learning only a single parameter.

Temperature Scaling

For a classifier with logits $\mathbf{z}(x)$, the softmax output is:

$$\hat{p}_i(x) = \frac{\exp(z_i(x))}{\sum_j \exp(z_j(x))}$$

Temperature scaling introduces a single parameter $T > 0$:

$$\hat{p}_i^{(T)}(x) = \frac{\exp(z_i(x)/T)}{\sum_j \exp(z_j(x)/T)}$$

Effect of T :

- $T > 1$: Softens probabilities (less confident)—dividing by $T > 1$ makes logit differences smaller, so probabilities become more uniform
- $T < 1$: Sharpens probabilities (more confident)—probabilities become more concentrated on the highest logit
- $T = 1$: Original probabilities

Fitting: Find T that minimises negative log-likelihood on a held-out calibration set.

Temperature scaling does not change predictions (same arg max), only confidence levels.

Why it works: Modern neural networks are typically overconfident. Temperature scaling with $T > 1$ “cools” the distribution, spreading probability mass more evenly and reducing overconfidence. The single parameter is learned to match predicted confidence to actual accuracy.

Why only one parameter? Remarkably, a single scalar parameter often suffices. This is because neural networks tend to be miscalibrated in a systematic way (uniformly overconfident), which can be corrected by a uniform rescaling of logits.

134.4 Platt Scaling

A more flexible calibration method, originally developed for SVMs.

Platt Scaling

For binary classification, transform the model output $f(x)$ (e.g., SVM score or logit) to a probability:

$$\hat{p}(y = 1 | x) = \sigma(Af(x) + B) = \frac{1}{1 + \exp(-(Af(x) + B))}$$

Parameters A and B are fit by maximum likelihood on a calibration set.

Generalisation to multiclass: Fit a logistic regression on the logit vector.

Comparison with temperature scaling:

- Temperature scaling: one parameter, only rescales logits
- Platt scaling: two parameters (for binary), can also shift the decision boundary
- Platt scaling is more flexible but risks overfitting on small calibration sets

134.5 Calibration in Regression

For regression, calibration means prediction intervals achieve nominal coverage.

Regression Calibration

A regression model is calibrated if its $p\%$ prediction intervals contain the true value approximately $p\%$ of the time, for all p .

Checking calibration:

1. For each test point, compute the predicted CDF value of the true outcome: $F_x(y)$ where F_x is the predicted CDF at input x
2. If calibrated, these values should be uniform on $[0, 1]$
3. Plot the empirical CDF of these values against the diagonal (probability integral transform)

Recalibration: If prediction intervals are too narrow/wide, scale the predicted variance.

Intuition: If I predict $Y \sim \mathcal{N}(\mu, \sigma^2)$, I am claiming that Y has a 50% chance of being below μ and a 95% chance of being within $\mu \pm 1.96\sigma$. If I am calibrated, these should be true empirically. The probability integral transform checks this by mapping each outcome to its predicted percentile—if calibrated, these percentiles should be uniform.

135 Practical Guidance

Section Summary: When to Use What

- **GPs:** Small-medium data, need interpretable uncertainty, Bayesian optimisation
- **Conformal:** Any model, need coverage guarantees, distribution-free setting
- **Deep ensembles:** Large data, neural networks, practical and effective
- **Always:** Check calibration; recalibrate if necessary

135.1 Method Selection

Decision Framework

Dataset size:

- $n < 1,000$: GPs are ideal; exact inference is tractable
- $1,000 < n < 100,000$: Sparse GPs, or conformal on any model
- $n > 100,000$: Deep ensembles, MC Dropout, or conformal prediction

Input dimension:

- Low ($D < 20$): GPs work well
- Medium ($20 < D < 100$): GPs possible with ARD, but consider neural methods
- High ($D > 100$): Neural networks typically outperform GPs

Uncertainty requirements:

- Need full posterior: GPs or variational BNNs
- Need prediction intervals: Conformal prediction or GPs
- Need point estimate + rough uncertainty: Deep ensembles or MC Dropout

Distributional assumptions:

- Can assume Gaussian: GPs provide stronger guarantees
- Distribution-free needed: Conformal prediction

135.2 Computational Considerations

Method	Training	Prediction	Memory
Exact GP	$O(n^3)$	$O(n^2)$ per point	$O(n^2)$
Sparse GP	$O(nm^2)$	$O(m^2)$ per point	$O(nm)$
Conformal	Model training + $O(n)$	Model + $O(1)$	$O(n)$ scores
Deep ensemble	$M \times$ model training	$M \times$ forward pass	$M \times$ model
MC Dropout	Model training	$T \times$ forward pass	Model

135.3 Validating Uncertainty Estimates

Uncertainty Validation Checklist

1. **Check calibration:** Plot reliability diagrams, compute ECE
2. **Check coverage:** Do prediction intervals achieve nominal coverage?
3. **Check sharpness:** Among calibrated methods, prefer tighter intervals
4. **Check out-of-distribution behaviour:** Does uncertainty increase for unusual inputs?
5. **Stratify by subgroups:** Is coverage consistent across different regions of input space?

Sharpness vs calibration tradeoff: A trivially calibrated predictor could give very wide intervals that always achieve coverage. We want intervals that are both calibrated *and* as tight as possible. Among calibrated methods, prefer the one with narrower intervals.

Out-of-distribution behaviour: A key test for uncertainty methods is whether they appropriately increase uncertainty for inputs unlike the training data. A good uncertainty method should say “I don’t know” when presented with genuinely novel inputs.

NB!

Common pitfalls:

- Assuming model uncertainty is calibrated without checking
- Conflating marginal and conditional coverage
- Using calibration set that overlaps with test set
- Ignoring that recalibration can harm out-of-distribution detection

On the last point: If you recalibrate a model to have good coverage on in-distribution data, you might inadvertently reduce its ability to flag out-of-distribution inputs as uncertain. The model might become “confidently wrong” on unusual inputs. Always validate both in-distribution calibration and out-of-distribution uncertainty behaviour.

136 Summary

Key Concepts from Week 9

1. **Types of uncertainty:** Epistemic (reducible with data) vs aleatoric (irreducible noise)
2. **Gaussian Processes:**
 - Distributions over functions, specified by mean and kernel
 - Any finite set of function values is jointly Gaussian
 - Posterior mean = KRR prediction; posterior variance provides calibrated uncertainty
 - Variance increases away from training data (desirable property)
 - $O(n^3)$ exact inference; sparse methods scale to larger n
3. **Kernels:**
 - Encode prior assumptions (smoothness, periodicity, length-scale)
 - RBF (infinitely smooth), Matérn (tunable smoothness), periodic
 - Combine via sum (independent components) or product (interactions)
 - Hyperparameters learned via marginal likelihood
4. **Conformal prediction:**
 - Distribution-free prediction sets with coverage guarantees
 - Works with any model; requires only exchangeability
 - Split conformal: train, calibrate, construct sets using quantile of scores
 - Provides marginal, not conditional, coverage
5. **Bayesian Neural Networks:**
 - Distributions over weights; exact posterior intractable
 - MC Dropout: dropout at test time approximates Bayesian inference
 - Deep ensembles: often work better than more principled approximations
6. **Calibration:**
 - Predicted probabilities should match empirical frequencies
 - Reliability diagrams and ECE for evaluation
 - Temperature scaling: simple, effective post-hoc recalibration

Practical Summary

Small data, need uncertainty: Use GPs

Any model, need coverage guarantee: Use conformal prediction

Large data, neural networks: Use deep ensembles or MC Dropout

Always: Validate calibration on held-out data; recalibrate if necessary

137 Overview

Chapter Summary

This chapter introduces the foundations of neural networks—from biological inspiration to practical training. We begin with the perceptron, prove its convergence for linearly separable data, and demonstrate its fundamental limitations. Multi-layer perceptrons (MLPs) overcome these limitations through learned feature representations. Key concepts include:

- **Perceptrons:** Linear classifiers with provable convergence guarantees
- **Linear separability:** The boundary of what single-layer networks can learn
- **MLPs:** Composing simple functions to build universal approximators
- **Backpropagation:** Efficient gradient computation via the chain rule
- **Training:** Loss functions, optimisers, initialisation, and regularisation

Neural networks represent a fundamental paradigm shift in machine learning. In the models we have studied previously—linear regression, logistic regression, kernel methods—we either work with features directly or apply hand-crafted transformations (polynomial features, radial basis functions). The key limitation is that these transformations are *fixed*: we choose them based on domain knowledge before seeing any data, then learn only the weights that combine these features.

Neural networks take a radically different approach: they learn the feature transformations themselves. This means the model can discover which patterns in the data are relevant, rather than relying on our ability to anticipate them.

Neural Networks in Context

Neural networks extend linear models by:

1. **Learning feature representations** $\phi(x)$ rather than hand-crafting them
2. **Composing simple functions** to build complex mappings
3. **Using non-linear activations** to escape the limitations of linear models

The transition from $f(x) = w^\top x$ to $f(x) = w^\top \phi(x; \theta)$ where ϕ is *learned* represents a fundamental shift in machine learning.

To understand what this means concretely, consider image classification. A linear model applied to raw pixels would need to learn that “cat-ness” is some weighted combination of individual pixel values. This is hopelessly naive—the same cat in a different position would have completely different pixel values. What we need are features like “presence of pointed ears,” “whisker-like patterns,” or “feline body shape.” Hand-crafting such features is impractical for complex tasks. A neural network learns a hierarchy of features: early layers might detect edges, middle layers combine edges into textures and shapes, and later layers recognise high-level concepts like “cat” or “dog.” Crucially, *all of these features emerge from training on data*—we do not specify them in advance.

138 Biological Motivation

Section Summary

Artificial neural networks draw inspiration from biological neurons, though the analogy is loose. Understanding the biological motivation provides context for design choices, while recognising the differences prevents over-interpreting “brain-like” claims.

The term “neural network” comes from early attempts to model computation in biological brains. While modern deep learning has diverged substantially from neuroscience, the historical connection explains much of the terminology and provides useful intuition.

138.1 Neurons and Synapses

The brain contains approximately 10^{11} neurons, each connected to thousands of others via synapses. A biological neuron operates through a cycle of signal reception, integration, and transmission:

1. **Receives signals** from other neurons via dendrites
2. **Integrates** these signals in the cell body (soma)
3. **Fires** an action potential down the axon if the integrated signal exceeds a threshold
4. **Transmits** the signal to downstream neurons via synaptic connections

The strength of synaptic connections varies—some are excitatory (encouraging the downstream neuron to fire), others inhibitory (discouraging firing)—and these strengths change with experience. This **synaptic plasticity** is believed to underlie learning and memory: connections that contribute to successful outcomes are strengthened, while others weaken.

138.2 The McCulloch-Pitts Neuron (1943)

The first computational model of a neuron was proposed by McCulloch and Pitts in 1943 (see Week 1). Their model abstracted the biological neuron as a binary threshold unit, capturing the essential computation of “weighted summation followed by thresholding.”

McCulloch-Pitts Neuron

A neuron j with n binary inputs x_1, \dots, x_n , weights w_1, \dots, w_n , and threshold θ computes:

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

McCulloch and Pitts showed that networks of such neurons could compute any Boolean function, establishing a connection between neural activity and logic.

What this model captures: The neuron receives inputs x_i (analogous to signals from other neurons), each weighted by w_i (analogous to synaptic strength). It sums these contributions and compares to a threshold θ . If the weighted sum is large enough, the neuron “fires” (outputs 1); otherwise it remains silent (outputs 0).

What this model lacks: The weights in McCulloch-Pitts neurons were *fixed*, not learned. The model described what neurons could compute, but not how brains could learn to compute it.

138.3 From Biology to Artificial Networks

NB!

Modern artificial neural networks differ substantially from biological neurons:

- **Continuous activations:** Real neurons communicate via spike trains (sequences of discrete pulses); artificial neurons use continuous values
- **Synchronous updates:** Biological neural activity is asynchronous and parallel; artificial networks update in discrete layers
- **Backpropagation:** There is no known biological mechanism for propagating error gradients backwards through synapses
- **Architecture:** Biological connectivity is sparse and irregular; artificial networks often use dense, structured connections

The “neural” in neural networks is more historical inspiration than literal description.

Despite these differences, the biological analogy remains useful for intuition:

- Weights \approx synaptic strengths (learned from experience)
- Activation functions \approx firing rate as a function of input current
- Layers \approx hierarchical processing in sensory cortex

But modern deep learning succeeds through mathematics and engineering, not biological fidelity. The field has developed its own principles that work well empirically, even where they diverge from neuroscience.

139 The Perceptron

Section Summary

The perceptron (Rosenblatt, 1957) was the first neural network that could *learn* from data. It implements a linear classifier with a simple update rule that provably converges for linearly separable data. Understanding the perceptron—its power and limitations—provides the foundation for understanding deeper networks.

The perceptron, introduced by Frank Rosenblatt in 1957, addressed the key limitation of McCulloch-Pitts neurons: it could *learn* its weights from examples. This was a revolutionary idea—a machine that could automatically adjust its behaviour based on experience.

The perceptron is a linear classifier: a non-probabilistic counterpart to logistic regression. Where logistic regression outputs probabilities via the sigmoid function, the perceptron outputs hard decisions via a step function.

139.1 Architecture

Perceptron Model

For input $x \in \mathbb{R}^d$ and parameters $w \in \mathbb{R}^d$, $b \in \mathbb{R}$:

Pre-activation (weighted sum):

$$z = w^\top x + b = \sum_{i=1}^d w_i x_i + b$$

Activation (threshold):

$$\hat{y} = \text{sign}(z) = \begin{cases} +1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases}$$

Compact notation: By augmenting x with a 1 (i.e., $\tilde{x} = [x; 1]$) and w with b (i.e., $\tilde{w} = [w; b]$), we can write:

$$\hat{y} = \text{sign}(\tilde{w}^\top \tilde{x})$$

Let us unpack this notation carefully:

- **Input** $x \in \mathbb{R}^d$: A vector of d features describing one example (e.g., pixel values, measurements, etc.)
- **Weight vector** $w \in \mathbb{R}^d$: One weight per feature, learned during training. Large positive w_i means feature i is strong evidence for class +1; large negative w_i means it is evidence for class -1.
- **Bias** $b \in \mathbb{R}$: Shifts the decision boundary. Without b , the decision boundary would always pass through the origin.
- **Pre-activation** z : The “score” for the positive class. Positive z predicts +1; negative z predicts -1.
- **Sign function**: Converts the continuous score into a discrete prediction.

The compact notation $\tilde{w}^\top \tilde{x}$ absorbs the bias into the weights by treating it as the weight for a constant “feature” that always equals 1. This simplifies the mathematics without changing the model.

The perceptron computes whether the input lies on the positive or negative side of a hyperplane defined by $w^\top x + b = 0$.

139.2 Geometric Interpretation

The perceptron implements a geometric concept: it partitions space using a hyperplane.

Hyperplane Classifier

The weight vector w defines a hyperplane in \mathbb{R}^d :

$$H = \{x \in \mathbb{R}^d : w^\top x + b = 0\}$$

Properties:

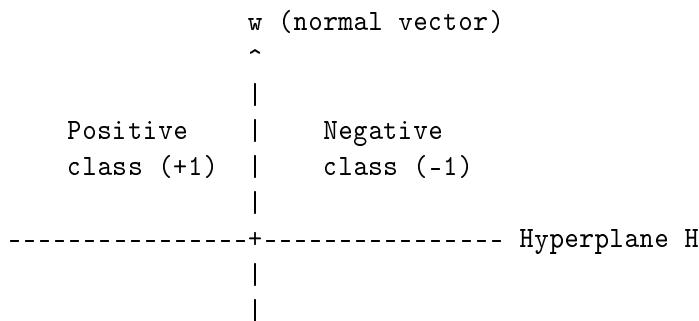
- w is the normal vector to the hyperplane (perpendicular to H)
- $b/\|w\|$ is the signed distance from the origin to H
- Points with $w^\top x + b > 0$ lie on the positive side; $w^\top x + b < 0$ on the negative side
- The signed distance from point x to H is $\frac{w^\top x + b}{\|w\|}$

Unpacking the geometry:

In two dimensions ($d = 2$), the hyperplane H is a line. Points on one side of the line are classified as $+1$; points on the other side as -1 . The weight vector w points “into” the positive region—it is perpendicular to the line and points toward class $+1$.

In three dimensions ($d = 3$), H is a plane. In higher dimensions, H is a $(d - 1)$ -dimensional subspace called a hyperplane.

The quantity $w^\top x + b$ measures how “deep” point x is into the positive region. The normalised version $(w^\top x + b)/\|w\|$ gives the actual geometric distance from x to the decision boundary.



The perceptron’s task is to find a hyperplane that separates positive from negative examples.

139.3 The Perceptron Learning Algorithm

Rosenblatt’s key contribution was not just the perceptron model, but a learning algorithm that could automatically find the weights.

Perceptron Learning Algorithm

Input: Training set $\{(x_1, y_1), \dots, (x_n, y_n)\}$ with $x_i \in \mathbb{R}^d$, $y_i \in \{-1, +1\}$

Initialisation: $w \leftarrow 0$, $b \leftarrow 0$

Repeat until convergence:

1. For each training example (x_i, y_i) :
2. Compute prediction: $\hat{y}_i = \text{sign}(w^\top x_i + b)$
3. If $\hat{y}_i \neq y_i$ (misclassification):
4. $w \leftarrow w + \eta y_i x_i$
5. $b \leftarrow b + \eta y_i$

Output: Classifier $f(x) = \text{sign}(w^\top x + b)$

Here $\eta > 0$ is the learning rate (often set to 1 for the basic algorithm).

Reading the algorithm:

- We initialise all weights to zero (the hyperplane is undefined, but the algorithm will quickly give it shape).
- We cycle through training examples. For each correctly classified example, we do nothing—no news is good news.
- When we misclassify an example (x_i, y_i) , we adjust the weights to make the correct answer more likely next time.
- The learning rate η controls how large each adjustment is. With $\eta = 1$, we get the classical perceptron algorithm.

Pseudocode:

```
def perceptron_train(X, y, max_iterations=1000, eta=1.0):
    n, d = X.shape
    w = np.zeros(d)
    b = 0.0

    for _ in range(max_iterations):
        errors = 0
        for i in range(n):
            y_hat = np.sign(np.dot(w, X[i]) + b)
            if y_hat != y[i]:
                w = w + eta * y[i] * X[i]
                b = b + eta * y[i]
                errors += 1
        if errors == 0:
            break # Converged

    return w, b
```

139.4 Understanding the Update Rule

Why does the update $w \leftarrow w + \eta y_i x_i$ make sense? Let us trace through the logic carefully.

Update Rule Intuition

Consider a misclassified point (x_i, y_i) where $y_i = +1$ but $w^\top x_i + b < 0$.

After the update $w' = w + \eta x_i$:

$$(w')^\top x_i = w^\top x_i + \eta \|x_i\|^2 > w^\top x_i$$

The new weight vector increases the score for x_i , pushing it toward correct classification.

Similarly, if $y_i = -1$ but $w^\top x_i + b > 0$, the update $w' = w - \eta x_i$ decreases the score for x_i .

Geometric view: The update rotates the hyperplane toward correctly classifying the misclassified point.

Detailed explanation: When we misclassify a positive example ($y_i = +1$), the current weights give it a negative score ($w^\top x_i < 0$). We want to increase this score. Adding ηx_i to w does exactly that: the new score is $(w + \eta x_i)^\top x_i = w^\top x_i + \eta \|x_i\|^2$. Since $\|x_i\|^2 > 0$, the score increases. Similarly, when we misclassify a negative example ($y_i = -1$), we subtract ηx_i from w , which decreases the score.

The factor y_i in the update rule $w \leftarrow w + \eta y_i x_i$ elegantly handles both cases: it is $+1$ for positive examples (add x_i) and -1 for negative examples (subtract x_i).

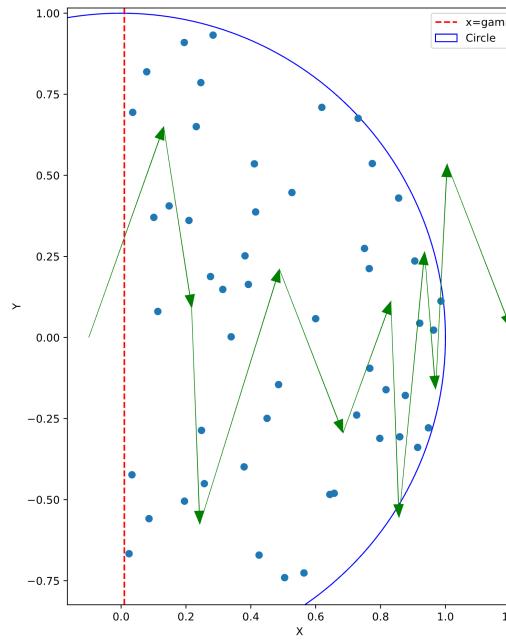


Figure 75: Perceptron learning: the decision boundary (dashed line) adjusts iteratively as misclassifications are corrected.

Connection to gradient descent: The perceptron update resembles gradient descent on logis-

tic regression's negative log-likelihood, but operates on single observations rather than batches and uses hard thresholds rather than probabilities. In fact, the perceptron update can be derived as a subgradient of the *hinge loss* (used in SVMs), though Rosenblatt discovered it through different reasoning.

139.5 The Perceptron Convergence Theorem

The perceptron algorithm is guaranteed to converge if the data is linearly separable. This was one of the first theoretical results about learning algorithms.

Perceptron Convergence Theorem

Assumptions:

1. The data is linearly separable: there exists $w^* \in \mathbb{R}^d$ with $\|w^*\| = 1$ and margin $\gamma > 0$ such that $y_i(w^{*\top} x_i) \geq \gamma$ for all i
2. All data points satisfy $\|x_i\| \leq R$

Conclusion: The perceptron algorithm makes at most $\left(\frac{R}{\gamma}\right)^2$ updates before finding a separating hyperplane.

Unpacking the assumptions:

- **w^* with $\|w^*\| = 1$:** A unit-length weight vector that correctly classifies all points. We normalise to unit length to make the margin well-defined.
- **Margin γ :** The minimum “depth” of any point into its correct region. All positive points have $w^{*\top} x_i \geq \gamma$; all negative points have $w^{*\top} x_i \leq -\gamma$. This measures how “easy” the separation is.
- **Radius R :** The maximum distance of any point from the origin. This bounds how “spread out” the data is.

The bound $(R/\gamma)^2$ tells us that convergence is fast when the margin is large relative to the data spread.

Proof Sketch

Let w^* be the optimal separator with $\|w^*\| = 1$ and margin γ . We track two quantities through the updates.

Progress toward w^* : After k updates (on examples $(x_{i_1}, y_{i_1}), \dots, (x_{i_k}, y_{i_k})$), with $\eta = 1$:

$$w^{(k)} = \sum_{j=1}^k y_{i_j} x_{i_j}$$

$$w^{(k)} \cdot w^* = \sum_{j=1}^k y_{i_j} (w^{*\top} x_{i_j}) \geq k\gamma$$

Each update increases $w^{(k)} \cdot w^*$ by at least γ .

Bounded growth: Since updates only occur on misclassified points (where $y_i(w^{\top} x_i) \leq 0$):

$$\|w^{(k)}\|^2 = \|w^{(k-1)}\|^2 + 2y_{i_k}(w^{(k-1)\top} x_{i_k}) + \|x_{i_k}\|^2 \leq \|w^{(k-1)}\|^2 + R^2$$

By induction: $\|w^{(k)}\|^2 \leq kR^2$.

Combining: By Cauchy-Schwarz:

$$k\gamma \leq w^{(k)} \cdot w^* \leq \|w^{(k)}\| \cdot \|w^*\| = \|w^{(k)}\| \leq \sqrt{k}R$$

Therefore $k \leq (R/\gamma)^2$. □

Walking through the proof:

The proof bounds k (the number of updates) by tracking two quantities:

1. **Alignment with w^* :** Each update on a misclassified point adds $y_{i_j} x_{i_j}$ to our weight vector. When we dot this with the optimal w^* , we get $y_{i_j} (w^{*\top} x_{i_j})$, which by assumption is at least γ . So after k updates, $w^{(k)} \cdot w^* \geq k\gamma$.
2. **Magnitude of $w^{(k)}$:** We cannot have $w^{(k)}$ growing too fast, because updates only happen on misclassified points. The key insight is that $y_{i_k} (w^{(k-1)\top} x_{i_k}) \leq 0$ for misclassified points, so the cross-term in $\|w^{(k)}\|^2$ is non-positive. This gives $\|w^{(k)}\|^2 \leq kR^2$.
3. **The contradiction:** Cauchy-Schwarz says $w^{(k)} \cdot w^* \leq \|w^{(k)}\|$. Combining with step 1: $k\gamma \leq \|w^{(k)}\| \leq \sqrt{k}R$. Squaring: $k^2\gamma^2 \leq kR^2$, so $k \leq (R/\gamma)^2$.

Convergence Bound Interpretation

The bound $k \leq (R/\gamma)^2$ reveals:

- **Larger margin γ :** Faster convergence—well-separated data is easier to learn
- **Larger radius R :** Slower convergence—spread-out data requires more adjustments
- **Dimension-independent:** The bound depends only on R/γ , not on d

This is an early example of a margin-based generalisation bound—ideas that later became central to SVMs and statistical learning theory.

140 Limitations of the Perceptron

Section Summary

The perceptron can only learn linearly separable functions. The XOR problem demonstrates this limitation geometrically. More generally, single-layer networks cannot represent functions requiring nonlinear decision boundaries. These limitations, highlighted by Minsky & Papert (1969), shaped the field's subsequent development.

The perceptron convergence theorem has a critical assumption: linear separability. When this assumption fails, the perceptron algorithm has no guarantee of success—in fact, it will cycle indefinitely without converging.

140.1 Linear Separability Requirement

Linear Separability

A dataset $\{(x_i, y_i)\}$ with $y_i \in \{-1, +1\}$ is **linearly separable** if there exists a hyperplane $w^\top x + b = 0$ such that:

$$y_i(w^\top x_i + b) > 0 \quad \text{for all } i$$

Equivalently: all positive examples lie strictly on one side of the hyperplane, and all negative examples on the other.

This is a strong assumption. In two dimensions, it means the classes can be separated by a straight line. In three dimensions, by a plane. Many real-world problems are not linearly separable in their original feature space.

The perceptron convergence theorem guarantees finding a separator *only if one exists*. If the data is not linearly separable, the algorithm will cycle indefinitely without converging—it will keep finding misclassified points and updating, but never reach a state where all points are correctly classified.

140.2 The XOR Problem

The XOR (exclusive or) function is the canonical example of a non-linearly-separable problem.

XOR Function

x_1	x_2	$x_1 \oplus x_2$
0	0	0 (negative)
0	1	1 (positive)
1	0	1 (positive)
1	1	0 (negative)

Geometric view: In \mathbb{R}^2 , the positive examples $(0, 1)$ and $(1, 0)$ occupy opposite corners of the unit square from the negative examples $(0, 0)$ and $(1, 1)$.

XOR is “exclusive or”: it returns 1 when exactly one input is 1, and 0 when both inputs are the same.

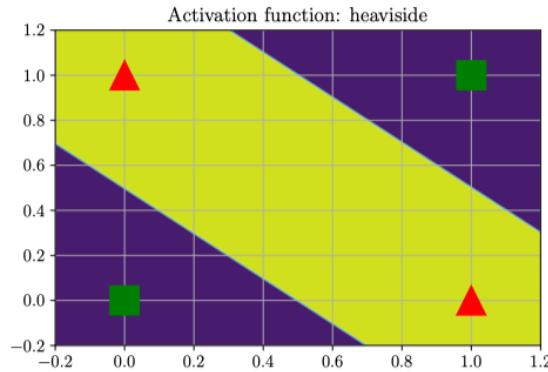


Figure 76: XOR is not linearly separable—no single line can separate the classes.

Why no hyperplane works: Consider trying to draw a line separating the positive examples from the negative ones:

- Any line separating $(0, 0)$ from $(0, 1)$ must pass between them (vertically or with positive slope on the left).
- Any line separating $(1, 1)$ from $(1, 0)$ must pass between them (vertically or with positive slope on the right).
- But a single straight line cannot simultaneously separate $(0, 0)$ from $(0, 1)$ and $(1, 1)$ from $(1, 0)$ while putting $(0, 1)$ and $(1, 0)$ on the same side.

Another way to see this: the positive examples are “diagonal” from each other, as are the negative examples. No single line can separate two diagonally opposite corners from the other two.

What the Perceptron Cannot Compute

A single perceptron can compute:

- AND: $x_1 \wedge x_2$ (threshold at 1.5 with weights $w_1 = w_2 = 1$)
- OR: $x_1 \vee x_2$ (threshold at 0.5 with weights $w_1 = w_2 = 1$)
- NOT: $\neg x_1$ (threshold at 0.5 with weight $w_1 = -1$)
- Any linearly separable Boolean function

A single perceptron **cannot** compute:

- XOR: $x_1 \oplus x_2$
- XNOR: $\neg(x_1 \oplus x_2)$
- Any non-linearly-separable function

Verifying AND is linearly separable: With $w_1 = w_2 = 1$ and threshold (bias) $b = -1.5$, the perceptron outputs 1 when $x_1 + x_2 - 1.5 \geq 0$, i.e., when $x_1 + x_2 \geq 1.5$. This is only true when both $x_1 = 1$ and $x_2 = 1$.

Why XOR is different: XOR would require outputting 1 when $x_1 + x_2 = 1$ exactly, but 0 when $x_1 + x_2 = 0$ or $x_1 + x_2 = 2$. No single threshold on a weighted sum can achieve this.

140.3 No Margin Maximisation

Even when data is linearly separable, infinitely many hyperplanes correctly classify all points.

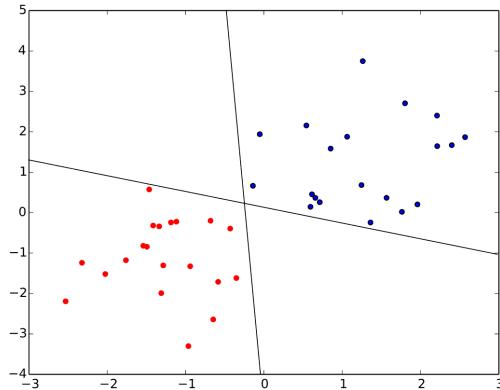


Figure 77: Multiple valid decision boundaries exist for linearly separable data.

The perceptron has no mechanism to prefer one over another—it simply finds *some* separator, which depends on the order of training examples and initialisation. This contrasts with Support Vector Machines, which explicitly maximise the margin (see Week 6 on kernels).

From a generalisation perspective, some hyperplanes are better than others. A hyperplane that barely separates the classes (small margin) is more likely to misclassify new points than one with a large margin. The perceptron provides no such guarantee.

NB!

These limitations, publicised by Minsky & Papert (1969) in their book *Perceptrons*, triggered the first “AI winter.” Their mathematical analysis showed that single-layer networks could not learn many important functions. Interest revived in the 1980s with multi-layer networks and backpropagation—architectures that *were* known in 1969, but for which no efficient training algorithm was widely available.

140.4 Solutions to the XOR Problem

The XOR problem admits several solutions, each historically significant:

Approaches to XOR

- 1. Feature engineering:** Map inputs to a space where they become linearly separable. Adding x_1x_2 as a feature:

x_1	x_2	x_1x_2	y
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Now the hyperplane $x_1 + x_2 - 2x_1x_2 = 0.5$ separates the classes.

- 2. Multi-layer networks:** Add a hidden layer that learns intermediate representations. $\text{XOR} = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$ decomposes into linearly separable sub-problems.
- 3. Kernel methods:** Implicitly map to high-dimensional feature spaces (see Week 6).

Why feature engineering works: In the augmented space (x_1, x_2, x_1x_2) :

- $(0, 0) \mapsto (0, 0, 0)$ — class 0
- $(0, 1) \mapsto (0, 1, 0)$ — class 1
- $(1, 0) \mapsto (1, 0, 0)$ — class 1
- $(1, 1) \mapsto (1, 1, 1)$ — class 0

The class 1 points have $x_1x_2 = 0$ and $x_1 + x_2 = 1$; the class 0 points have either $x_1 + x_2 = 0$ or $x_1x_2 = 1$. A linear separator exists in this 3D space.

Multi-layer networks automate the feature engineering—the hidden layers learn representations that make the problem linearly separable in the final layer. This is the key insight that revived neural networks.

141 Multi-Layer Perceptrons (MLPs)

Section Summary

Multi-layer perceptrons overcome single-layer limitations by stacking layers of neurons with nonlinear activations. Each layer transforms its input, learning increasingly abstract representations. With sufficient width, a single hidden layer can approximate any continuous function (universal approximation), though deeper networks often learn more efficiently.

The solution to the perceptron's limitations is deceptively simple: add more layers. A multi-layer perceptron (MLP) consists of an input layer, one or more hidden layers, and an output layer. Each hidden layer transforms its input through a linear operation followed by a nonlinear activation.

141.1 From Fixed to Learned Features

From Fixed to Learned Features

Traditional approach (linear models):

$$f(x; w) = w^\top \phi(x)$$

where $\phi(x)$ is a fixed, hand-crafted feature transformation.

Neural network approach:

$$f(x; W_2, W_1) = W_2^\top \phi(W_1^\top x)$$

where ϕ is a nonlinear activation applied element-wise, and both W_1 and W_2 are **learned**.

Deep networks compose multiple such transformations:

$$f(x; \theta) = f_L \circ f_{L-1} \circ \cdots \circ f_1(x)$$

where $f_l(z) = \sigma(W_l z + b_l)$ applies an affine transformation followed by nonlinearity σ .

What this means: Instead of choosing features like “degree-2 polynomial” or “RBF with bandwidth σ ,” we let the network discover useful features. The first layer might learn to detect certain patterns; the second layer combines these patterns into more complex features; and so on. The key insight: instead of designing features by hand, let the network learn them from data.

141.2 Architecture and Notation

MLP Notation

Consider an MLP with L layers:

Layer l parameters:

- Weight matrix: $W^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$
- Bias vector: $b^{(l)} \in \mathbb{R}^{n_l}$
- n_l = number of neurons in layer l

Forward computation:

$$\begin{aligned} z^{(l)} &= W^{(l)} a^{(l-1)} + b^{(l)} && \text{(pre-activation)} \\ a^{(l)} &= \sigma(z^{(l)}) && \text{(activation)} \end{aligned}$$

where $a^{(0)} = x$ (input) and σ is the activation function.

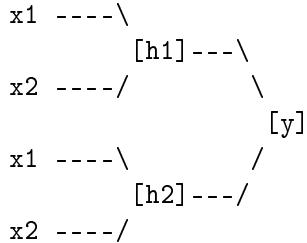
Network output: $\hat{y} = a^{(L)}$ (possibly with a different final activation, e.g., softmax for classification).

Reading the notation:

- $a^{(l)}$ is the activation (output) of layer l . The superscript (l) indicates the layer number.
- $z^{(l)}$ is the pre-activation: the result after the linear transformation but before the nonlinearity.

- $W^{(l)}$ maps from n_{l-1} inputs to n_l outputs. Row i of $W^{(l)}$ contains the weights for neuron i in layer l .
- $b^{(l)}$ contains one bias term per neuron in layer l .

Example: Two-layer network for XOR



Layer 0 (input): $a^0 = [x_1, x_2]$

Layer 1 (hidden): $z^1 = W^1 a^0 + b^1$, $a^1 = \text{sigma}(z^1)$

Layer 2 (output): $z^2 = W^2 a^1 + b^2$, $y = \text{sigma}(z^2)$

Here, the hidden layer has 2 neurons ($n_1 = 2$), the input has 2 features ($n_0 = 2$), and the output has 1 neuron ($n_2 = 1$). The weight matrix $W^{(1)}$ is 2×2 ; $W^{(2)}$ is 1×2 .

141.3 Why Non-Linearity is Essential

NB!

Without non-linear activations, composing linear layers produces a linear function:

$$f(x) = W_L W_{L-1} \cdots W_1 x = Mx$$

where $M = W_L W_{L-1} \cdots W_1$ is just another matrix.

Depth provides **no additional representational power** without non-linearity.

Why this matters: If W_1 is $n_1 \times d$, W_2 is $n_2 \times n_1$, etc., the product M is $n_L \times d$ —equivalent to a single linear layer. All the intermediate structure collapses.

Consider an extreme example: stacking 100 linear layers with 1000 neurons each. Without nonlinearities, this is equivalent to a single $n_L \times d$ matrix. We have 100 times the parameters but no more representational power.

The nonlinearity σ breaks this collapse, enabling the network to represent nonlinear functions. Each layer can now contribute new nonlinear structure, and depth becomes meaningful.

141.4 What MLPs Learn: A Geometric View

Each layer of an MLP performs a geometric transformation:

1. **Affine transformation** ($z = Wx + b$): Rotates, scales, shears, and translates
2. **Nonlinearity** ($a = \sigma(z)$): “Folds” or “bends” the space, creating nonlinear boundaries

For the XOR problem, the hidden layer learns to “unfold” the input space so that the two classes become linearly separable:

Input space:

$(0, 1) + (1, 0)$ After hidden layer transformation,

(0,0)	(1,1)	the + and - points become linearly separable
-	-	

Intuition: The hidden layer implements a nonlinear “warping” of space. Points that were not separable by a line in the original space become separable after the transformation. The output layer then simply draws a line in this transformed space.

141.5 Universal Approximation Theorem

Universal Approximation Theorem (Informal)

A feedforward network with a single hidden layer containing a finite number of neurons can approximate any continuous function on compact subsets of \mathbb{R}^n , to arbitrary precision, given:

- A suitable non-linear activation function (e.g., sigmoid, ReLU)
- Sufficiently many hidden neurons

What this says: Given any continuous function $g : \mathbb{R}^d \rightarrow \mathbb{R}$ and any desired accuracy $\epsilon > 0$, there exists an MLP with one hidden layer such that $|f(x) - g(x)| < \epsilon$ for all x in a bounded region.

What this does not say: It does not tell us how many hidden neurons we need (could be astronomically large), how to find the right weights (the optimisation problem could be hard), or whether the resulting network will generalise well.

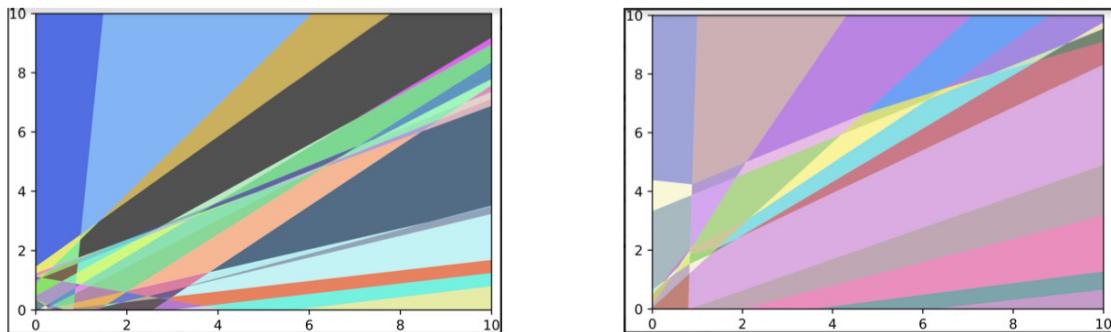


Figure 78: ReLU networks learn piecewise linear functions, partitioning input space into regions.

Intuition for ReLU networks: Each ReLU unit contributes a “hinge”—a point where the function changes slope. With enough hinges, any continuous function can be approximated by a piecewise linear function. More ReLU units mean finer approximation.

Universal Approximation: Theory vs Practice

The theorem says: “one hidden layer suffices in principle.”

But in practice:

- The required width may be exponentially large
- Deep networks often achieve the same approximation with fewer parameters
- Depth enables hierarchical representations that match data structure
- Optimisation is often easier with depth (when done right)

Universal approximation is an *existence* result, not a prescription for architecture.

141.6 Why Depth Matters

Representational Efficiency of Depth

Some functions can be represented exponentially more efficiently with deep networks:

Example: The function $f(x) = x_1 \cdot x_2 \cdot x_3 \cdots x_n$ (product of n inputs)

- **Shallow network:** Requires $\Omega(2^n)$ neurons to approximate
- **Deep network:** $O(n)$ neurons arranged in $O(\log n)$ layers suffice

The intuition: depth enables **reuse of intermediate computations**. Computing $((x_1 \cdot x_2) \cdot (x_3 \cdot x_4)) \cdot \dots$ shares subcomputations; a shallow network must enumerate all combinations.

A concrete example: To compute $x_1 \cdot x_2 \cdot x_3 \cdot x_4$:

- **Deep approach:** First compute $a = x_1 \cdot x_2$ and $b = x_3 \cdot x_4$, then compute $a \cdot b$. Total: 3 multiplications, depth 2.
- **Shallow approach:** Would need to somehow encode all 2^4 possible combinations of inputs and their contributions to the output.

More practically, deep networks learn **hierarchical representations**:

- Early layers: Low-level features (edges, textures)
- Middle layers: Parts and patterns
- Deep layers: Objects and concepts

This hierarchy matches the compositional structure of real-world data. Images are made of objects, which are made of parts, which are made of edges. Natural language has similar hierarchical structure.

142 Activation Functions

Section Summary

Activation functions introduce nonlinearity between layers. The choice of activation affects gradient flow, training dynamics, and what functions the network can represent. Modern practice favours ReLU and its variants for hidden layers, with task-specific activations (sigmoid, softmax) for outputs.

Activation functions are the nonlinear “glue” that makes neural networks powerful. Without them, as we have seen, depth provides no benefit. The choice of activation function has profound effects on training dynamics.

Common Activation Functions

Step function (original perceptron):

$$f(x) = \mathbb{I}(x \geq 0) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Binary output; no gradient for optimisation (derivative is 0 almost everywhere).

Sigmoid (logistic):

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \quad \sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Squashes output to $(0, 1)$; interpretable as probability. **Problem:** Saturates for large $|x|$, causing vanishing gradients.

Tanh (hyperbolic tangent):

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \quad \tanh'(x) = 1 - \tanh^2(x)$$

Squashes output to $(-1, 1)$; zero-centred (unlike sigmoid). Still saturates.

ReLU (Rectified Linear Unit):

$$\text{ReLU}(x) = \max(0, x), \quad \text{ReLU}'(x) = \mathbb{I}(x > 0)$$

Simple, efficient, non-saturating for positive inputs. **Problem:** “Dead neurons” when $x < 0$ forever.

Leaky ReLU:

$$\text{LeakyReLU}(x) = \max(\alpha x, x), \quad \text{where } \alpha \approx 0.01$$

Non-zero gradient for negative inputs prevents dead neurons.

ELU (Exponential Linear Unit):

$$\text{ELU}(x) = \begin{cases} x & x > 0 \\ \alpha(e^x - 1) & x \leq 0 \end{cases}$$

Smooth at zero; negative values help push mean activations toward zero.

GELU (Gaussian Error Linear Unit):

$$\text{GELU}(x) = x \cdot \Phi(x) \approx x \cdot \sigma(1.702x)$$

where Φ is the standard Gaussian CDF. Smooth, non-monotonic; used in Transformers.

Swish:

$$\text{Swish}(x) = x \cdot \sigma(x)$$

Smooth, non-monotonic; sometimes outperforms ReLU in deep networks.

Understanding each activation:

- **Sigmoid:** Historically important; outputs are bounded in $(0, 1)$, which is useful for probabilities. But it saturates: when $|x|$ is large, $\sigma'(x) \approx 0$, so gradients vanish.

- **Tanh:** Like sigmoid but centred at zero. Outputs in $(-1, 1)$. Same saturation problem.
- **ReLU:** The breakthrough activation for deep learning. For $x > 0$, ReLU is simply the identity, so gradients flow unchanged. For $x \leq 0$, ReLU outputs 0, which can cause “dead neurons” that never recover.
- **Leaky ReLU:** Fixes ReLU’s dead neuron problem by allowing a small gradient (α , typically 0.01) for negative inputs.
- **GELU/Swish:** Modern smooth activations that blend the benefits of ReLU (non-saturation for positive inputs) with smoothness at zero. Widely used in Transformers.

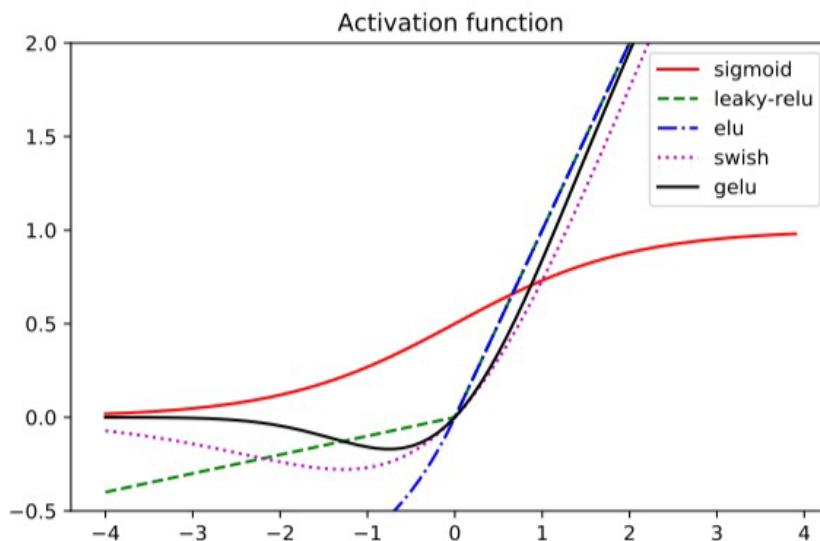


Figure 79: Comparison of common activation functions.

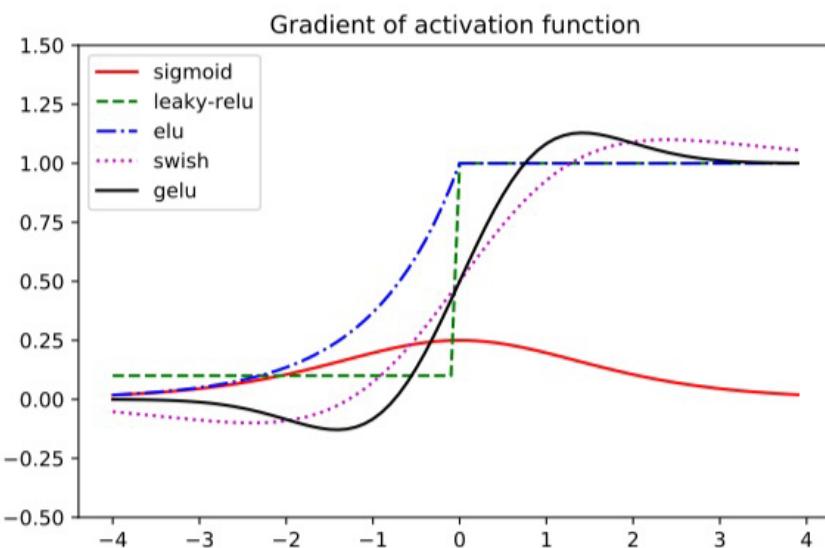


Figure 80: Activation function derivatives—crucial for gradient flow.

142.1 The Saturation Problem

Sigmoid Saturation

For sigmoid $\sigma(x)$:

- When $x \gg 0$: $\sigma(x) \approx 1$, $\sigma'(x) \approx 0$
- When $x \ll 0$: $\sigma(x) \approx 0$, $\sigma'(x) \approx 0$

In the **saturated regime**, gradients vanish:

$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial a} \cdot \sigma'(z) \approx 0$$

Learning stops because the gradient signal disappears.

Why saturation is problematic: During backpropagation, gradients multiply through layers. If $\sigma'(z) \approx 0$ at some layer, the gradient signal to earlier layers is near zero. Those layers cannot learn.

The maximum derivative of sigmoid is $\sigma'(0) = 0.25$. Even in the best case, each sigmoid layer reduces gradient magnitude by at least 75%. After several layers, gradients become negligibly small.

ReLU avoids saturation for positive inputs: $\text{ReLU}'(x) = 1$ for all $x > 0$. This enables training of much deeper networks.

NB!

ReLU's dead neuron problem: If a neuron's input is always negative, its gradient is always zero, and it can never recover. This typically happens when:

- Learning rate is too high, pushing weights to extreme values
- Poor initialisation places neurons in the dead zone

Leaky ReLU, ELU, and similar variants address this by maintaining a small gradient for negative inputs.

142.2 Output Layer Activations

The output layer activation depends on the task:

Output Activations by Task

Regression: No activation (or linear/identity). Output can be any real number.

Binary classification: Sigmoid. Output in $(0, 1)$ interpreted as $P(y = 1|x)$.

Multi-class classification: Softmax:

$$\text{softmax}(z)_k = \frac{e^{z_k}}{\sum_{j=1}^K e^{z_j}}$$

Outputs K positive values summing to 1, interpretable as class probabilities.

Activation Function Guidelines

- **Hidden layers:** ReLU is the default; Leaky ReLU or GELU if dead neurons are a concern
- **Output layer (regression):** Linear (no activation)
- **Output layer (binary classification):** Sigmoid
- **Output layer (multi-class):** Softmax
- **Avoid sigmoid/tanh in hidden layers** of deep networks (saturation)

143 Loss Functions

Section Summary

The loss function defines what the network should learn by measuring the discrepancy between predictions and targets. Mean squared error is standard for regression; cross-entropy for classification. The choice of loss affects gradients and training dynamics.

The loss function $\mathcal{L}(\hat{y}, y)$ measures how bad our predictions are. Training minimises this loss. The choice of loss function encodes our assumptions about the problem and affects how the network learns.

143.1 Mean Squared Error (Regression)

Mean Squared Error

For regression with predictions \hat{y}_i and targets y_i :

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Gradient:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_i} = \frac{2}{n} (\hat{y}_i - y_i)$$

Interpretation: MSE corresponds to maximum likelihood under Gaussian noise: if $y = f(x) + \epsilon$ with $\epsilon \sim \mathcal{N}(0, \sigma^2)$, minimising MSE is equivalent to maximising the likelihood.

Why MSE for regression?

1. **Probabilistic interpretation:** If we assume the true relationship is $y = f(x) + \epsilon$ where ϵ is Gaussian noise, the log-likelihood of observing y given prediction $f(x)$ is:

$$\log p(y|f(x)) = -\frac{(y - f(x))^2}{2\sigma^2} + \text{const}$$

Maximising this is equivalent to minimising $(y - f(x))^2$.

2. **Smooth gradients:** The gradient is proportional to the error. Large errors give large gradients, driving fast correction. Small errors give small gradients, enabling fine-tuning.

143.2 Cross-Entropy (Classification)

Binary Cross-Entropy

For binary classification with predicted probability $\hat{p}_i = \sigma(z_i)$ and label $y_i \in \{0, 1\}$:

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{n} \sum_{i=1}^n [y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i)]$$

Gradient with respect to logit z_i :

$$\frac{\partial \mathcal{L}}{\partial z_i} = \frac{1}{n} (\hat{p}_i - y_i)$$

This elegant form results from combining the sigmoid and log in the loss. Gradients do not vanish even when the sigmoid saturates, because the log “unsquashes” the output.

Unpacking binary cross-entropy:

- When $y_i = 1$: Loss is $-\log(\hat{p}_i)$. This is small when $\hat{p}_i \approx 1$ (correct confident prediction) and large when $\hat{p}_i \approx 0$ (wrong confident prediction).
- When $y_i = 0$: Loss is $-\log(1 - \hat{p}_i)$. This is small when $\hat{p}_i \approx 0$ and large when $\hat{p}_i \approx 1$.
- The loss penalises confident wrong predictions heavily (the $-\log$ goes to infinity as the probability goes to zero).

Why the gradient is so clean: The derivative $\frac{\partial \mathcal{L}}{\partial z_i} = \hat{p}_i - y_i$ is remarkably simple. For a positive example ($y_i = 1$), we push z_i up if $\hat{p}_i < 1$. For a negative example ($y_i = 0$), we push z_i down if $\hat{p}_i > 0$. The magnitude of the push is proportional to the error.

Multi-class Cross-Entropy

For K classes with predicted probabilities $\hat{p}_{i,k} = \text{softmax}(z_i)_k$ and one-hot label $y_{i,k}$:

$$\mathcal{L}_{\text{CE}} = -\frac{1}{n} \sum_{i=1}^n \sum_{k=1}^K y_{i,k} \log(\hat{p}_{i,k})$$

Since y is one-hot (only one class is 1, rest are 0), this simplifies to:

$$\mathcal{L}_{\text{CE}} = -\frac{1}{n} \sum_{i=1}^n \log(\hat{p}_{i,y_i})$$

Interpretation: Cross-entropy measures the “surprise” of the true labels under the predicted distribution. Minimising cross-entropy is equivalent to maximising the likelihood of the data under a categorical distribution.

Reading the formula: For each example i , we look at the probability assigned to the *correct* class y_i , take its log, and negate. If we are confident and correct ($\hat{p}_{i,y_i} \approx 1$), loss is near 0. If we are confident and wrong ($\hat{p}_{i,y_i} \approx 0$), loss is very large.

Why Cross-Entropy, Not MSE, for Classification?

- **Gradient magnitude:** With MSE and sigmoid output, gradients vanish when predictions are confident but wrong. Cross-entropy maintains strong gradients.
- **Probabilistic interpretation:** Cross-entropy corresponds to maximum likelihood for categorical outcomes.
- **Numerical stability:** Log-softmax can be computed stably; MSE with softmax is less stable.

The gradient problem with MSE for classification: With MSE loss and sigmoid output, the gradient is:

$$\frac{\partial \mathcal{L}}{\partial z} = (\hat{p} - y) \cdot \sigma'(z)$$

When z is large and wrong (e.g., $z = 10$ but $y = 0$), $\sigma(z) \approx 1$ and $\sigma'(z) \approx 0$. The gradient is tiny even though the prediction is badly wrong. Cross-entropy fixes this by using log, which cancels the squashing effect of sigmoid.

144 Backpropagation

Section Summary

Backpropagation is the algorithm for computing gradients in neural networks. It applies the chain rule systematically, computing gradients layer-by-layer from output to input. The key insight: reuse intermediate computations to achieve $O(W)$ complexity where W is the number of weights.

Backpropagation is the workhorse algorithm of deep learning. It efficiently computes how the loss changes with respect to every parameter in the network, enabling gradient-based optimisation.

144.1 The Credit Assignment Problem

Training a neural network requires computing $\nabla_{\theta}\mathcal{L}$: how does the loss change with respect to each parameter?

In deep networks, each parameter contributes to the loss through a long chain of transformations. The **credit assignment problem** asks: which parameters are “responsible” for the error?

Consider a 10-layer network with a prediction error. The error could be due to problems in any layer. Parameters in layer 1 affect the loss through 9 subsequent transformations. How do we attribute “blame” correctly?

Backpropagation answers this by propagating error signals backward through the network, attributing “credit” or “blame” to each parameter proportional to its gradient.

144.2 Chain Rule Review

Chain Rule for Composition

For scalar functions $f : \mathbb{R} \rightarrow \mathbb{R}$ and $g : \mathbb{R} \rightarrow \mathbb{R}$:

$$\frac{d}{dx} f(g(x)) = f'(g(x)) \cdot g'(x)$$

For vector functions $f : \mathbb{R}^m \rightarrow \mathbb{R}$ and $g : \mathbb{R}^n \rightarrow \mathbb{R}^m$:

$$\frac{\partial f}{\partial x_j} = \sum_{i=1}^m \frac{\partial f}{\partial g_i} \cdot \frac{\partial g_i}{\partial x_j}$$

In matrix form with Jacobians:

$$\nabla_x f(g(x)) = J_g^\top \nabla_g f$$

where $J_g \in \mathbb{R}^{m \times n}$ has entries $(J_g)_{ij} = \frac{\partial g_i}{\partial x_j}$.

The key insight: The chain rule tells us how to decompose the derivative of a composition. If $h = f \circ g$, then $h'(x) = f'(g(x)) \cdot g'(x)$. We can compute h' by first computing g' and f' separately, then multiplying.

For neural networks, we have many composed functions. The chain rule lets us decompose the gradient computation into local computations at each layer.

144.3 Forward and Backward Passes

Backpropagation via Chain Rule

For a network $\hat{y} = f_L \circ f_{L-1} \circ \dots \circ f_1(x)$:

Forward pass: Compute and store activations:

$$\begin{aligned} a^{(0)} &= x \\ z^{(l)} &= W^{(l)} a^{(l-1)} + b^{(l)} \\ a^{(l)} &= \sigma(z^{(l)}) \end{aligned}$$

Backward pass: Compute gradients from output to input:

$$\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial z^{(L)}} \quad (\text{output layer gradient})$$

For $l = L-1, \dots, 1$:

$$\delta^{(l)} = (W^{(l+1)})^\top \delta^{(l+1)} \odot \sigma'(z^{(l)})$$

where \odot denotes element-wise multiplication.

Parameter gradients:

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \delta^{(l)} (a^{(l-1)})^\top, \quad \frac{\partial \mathcal{L}}{\partial b^{(l)}} = \delta^{(l)}$$

Understanding the backward pass:

The quantity $\delta^{(l)} = \frac{\partial \mathcal{L}}{\partial z^{(l)}}$ is the “error signal” at layer l —how sensitive the loss is to changes in the pre-activations.

- $\delta^{(L)}$: Computed directly from the loss function. For cross-entropy with softmax, this is simply $\hat{p} - y$.
- $\delta^{(l)} = (W^{(l+1)})^\top \delta^{(l+1)} \odot \sigma'(z^{(l)})$: The error from layer $l + 1$ is propagated back through the weights $W^{(l+1)}$ and scaled by the activation derivative $\sigma'(z^{(l)})$.
- **Why $(W^{(l+1)})^\top$** : During the forward pass, $z^{(l+1)} = W^{(l+1)}a^{(l)}$. The Jacobian of this with respect to $a^{(l)}$ is $W^{(l+1)}$. In the backward pass, we multiply by the transpose.
- **Why $\odot \sigma'(z^{(l)})$** : The activation $a^{(l)} = \sigma(z^{(l)})$ introduces element-wise nonlinearity. Its Jacobian is diagonal with entries $\sigma'(z_i^{(l)})$.

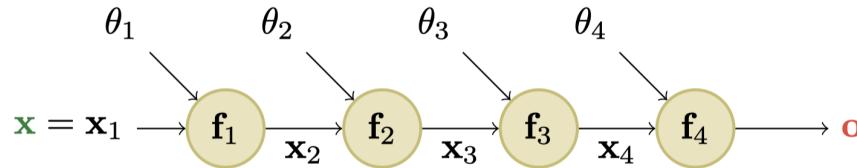


Figure 81: Backpropagation computes gradients layer by layer using the chain rule.

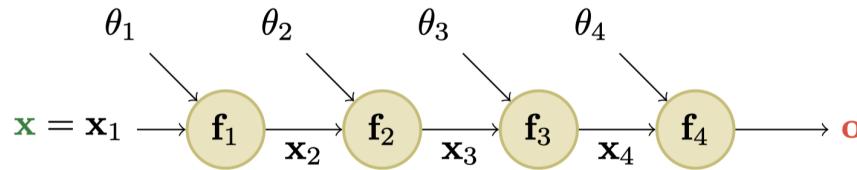


Figure 82: MLP structure with alternating linear and non-linear layers.

The gradient for each layer reuses computations from later layers. This is the key efficiency: we do not recompute the chain from scratch for each parameter.

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \theta_3} &= \frac{\partial \mathcal{L}}{\partial x_4} \cdot \frac{\partial x_4}{\partial \theta_3} \\ \frac{\partial \mathcal{L}}{\partial \theta_2} &= \frac{\partial \mathcal{L}}{\partial x_4} \cdot \frac{\partial x_4}{\partial x_3} \cdot \frac{\partial x_3}{\partial \theta_2} \\ \frac{\partial \mathcal{L}}{\partial \theta_1} &= \frac{\partial \mathcal{L}}{\partial x_4} \cdot \frac{\partial x_4}{\partial x_3} \cdot \frac{\partial x_3}{\partial x_2} \cdot \frac{\partial x_2}{\partial \theta_1}\end{aligned}$$

Notice how $\frac{\partial \mathcal{L}}{\partial x_4}$ and $\frac{\partial \mathcal{L}}{\partial x_4} \cdot \frac{\partial x_4}{\partial x_3}$ are reused across multiple parameter gradients.

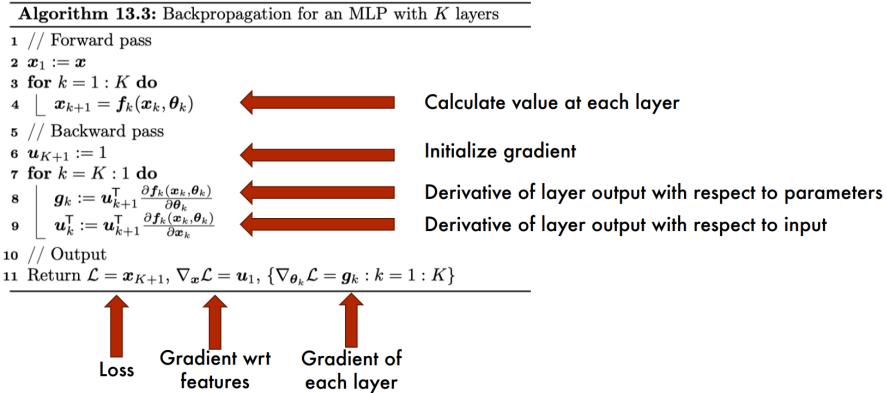


Figure 83: Backpropagation algorithm: gradients are computed iteratively from output to input.

144.4 Worked Example: Two-Layer Network

Backpropagation Example

Consider a network with:

- Input: $x \in \mathbb{R}^2$
- Hidden layer: 2 neurons, ReLU activation
- Output: 1 neuron, sigmoid activation, binary cross-entropy loss

Forward pass:

$$\begin{aligned} z^{(1)} &= W^{(1)}x + b^{(1)} \in \mathbb{R}^2 \\ a^{(1)} &= \text{ReLU}(z^{(1)}) \in \mathbb{R}^2 \\ z^{(2)} &= W^{(2)}a^{(1)} + b^{(2)} \in \mathbb{R} \\ \hat{y} &= \sigma(z^{(2)}) \in (0, 1) \\ \mathcal{L} &= -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})] \end{aligned}$$

Backward pass:

Step 1: Output layer gradient (combining sigmoid and cross-entropy):

$$\delta^{(2)} = \frac{\partial \mathcal{L}}{\partial z^{(2)}} = \hat{y} - y$$

Step 2: Gradients for $W^{(2)}, b^{(2)}$:

$$\frac{\partial \mathcal{L}}{\partial W^{(2)}} = \delta^{(2)}(a^{(1)})^\top, \quad \frac{\partial \mathcal{L}}{\partial b^{(2)}} = \delta^{(2)}$$

Step 3: Propagate to hidden layer:

$$\delta^{(1)} = (W^{(2)})^\top \delta^{(2)} \odot \mathbb{I}(z^{(1)} > 0)$$

Step 4: Gradients for $W^{(1)}, b^{(1)}$:

$$\frac{\partial \mathcal{L}}{\partial W^{(1)}} = \delta^{(1)}x^\top, \quad \frac{\partial \mathcal{L}}{\partial b^{(1)}} = \delta^{(1)}$$

Numerical example: Let $x = [1, 0.5]^\top$, $y = 1$, and suppose:

$$\begin{aligned} W^{(1)} &= \begin{bmatrix} 0.5 & 0.3 \\ -0.2 & 0.8 \end{bmatrix}, & b^{(1)} &= \begin{bmatrix} 0.1 \\ -0.1 \end{bmatrix} \\ W^{(2)} &= [0.6, 0.4], & b^{(2)} &= 0.2 \end{aligned}$$

Forward pass:

$$\begin{aligned} z^{(1)} &= \begin{bmatrix} 0.5 \cdot 1 + 0.3 \cdot 0.5 + 0.1 \\ -0.2 \cdot 1 + 0.8 \cdot 0.5 - 0.1 \end{bmatrix} = \begin{bmatrix} 0.75 \\ 0.1 \end{bmatrix} \\ a^{(1)} &= \text{ReLU}(z^{(1)}) = \begin{bmatrix} 0.75 \\ 0.1 \end{bmatrix} \quad (\text{both positive, so unchanged}) \\ z^{(2)} &= 0.6 \cdot 0.75 + 0.4 \cdot 0.1 + 0.2 = 0.69 \\ \hat{y} &= \sigma(0.69) \approx 0.666 \end{aligned}$$

Backward pass:

$$\begin{aligned} \delta^{(2)} &= 0.666 - 1 = -0.334 \quad (\text{we under-predicted; need to increase } z^{(2)}) \\ \frac{\partial \mathcal{L}}{\partial W^{(2)}} &= -0.334 \cdot [0.75, 0.1] = [-0.251, -0.033] \\ \delta^{(1)} &= \begin{bmatrix} 0.6 \\ 0.4 \end{bmatrix} \cdot (-0.334) \odot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -0.200 \\ -0.134 \end{bmatrix} \\ \frac{\partial \mathcal{L}}{\partial W^{(1)}} &= \begin{bmatrix} -0.200 \\ -0.134 \end{bmatrix} [1, 0.5] = \begin{bmatrix} -0.200 & -0.100 \\ -0.134 & -0.067 \end{bmatrix} \end{aligned}$$

Note that all gradients are negative, meaning we should increase all weights to increase the output toward the correct label $y = 1$.

144.5 Computational Graph Perspective

Modern frameworks represent computations as directed acyclic graphs (DAGs):

- **Nodes:** Operations (add, multiply, activation)
- **Edges:** Data flow (tensors)

Backpropagation traverses this graph in reverse topological order, applying the chain rule at each node. This generalises naturally to arbitrary architectures beyond simple feedforward networks.

144.6 Automatic Differentiation

Modern frameworks (PyTorch, TensorFlow, JAX) use **automatic differentiation**: decomposing functions into atomic operations with known derivatives, then applying chain rule automatically.

Differentiation Rules for Composition

- **Addition:** $\nabla_x(f + g) = \nabla_x f + \nabla_x g$
- **Multiplication:** $\nabla_x(f \cdot g) = f \cdot \nabla_x g + g \cdot \nabla_x f$
- **Composition:** $\nabla_x f(g(x)) = \nabla_u f(u)|_{u=g(x)} \cdot \nabla_x g(x)$

The key distinction from symbolic differentiation: autodiff operates on *evaluated* computation graphs, not symbolic expressions. It computes numerical gradients without forming explicit derivative expressions.

Forward vs reverse mode: Autodiff can be implemented in two modes:

- **Forward mode:** Computes $\frac{\partial \text{output}}{\partial \text{one input}}$ efficiently. Good when there are few inputs and many outputs.

- **Reverse mode:** Computes $\frac{\partial \text{one output}}{\partial \text{all inputs}}$ efficiently. Good when there is one output (the loss) and many inputs (parameters). This is backpropagation.

Neural networks have one loss and millions of parameters, so reverse mode (backprop) is the right choice.

145 Training Neural Networks

Section Summary

Training neural networks requires choosing an optimiser, learning rate, and batch size. Mini-batch stochastic gradient descent balances computational efficiency with gradient variance. Adam adapts learning rates per-parameter and is the default choice for most applications.

Training a neural network means finding parameters θ that minimise the loss. This is done via gradient descent (or variants), using gradients computed by backpropagation.

145.1 Gradient Descent

Gradient Descent Variants

Batch gradient descent: Use all n examples per update:

$$\theta \leftarrow \theta - \eta \cdot \frac{1}{n} \sum_{i=1}^n \nabla_{\theta} \ell(y_i, f(x_i; \theta))$$

Low variance but expensive per update.

Stochastic gradient descent (SGD): Use one example per update:

$$\theta \leftarrow \theta - \eta \cdot \nabla_{\theta} \ell(y_i, f(x_i; \theta))$$

Cheap per update but high variance.

Mini-batch SGD: Use B examples per update:

$$\theta \leftarrow \theta - \frac{\eta}{B} \sum_{b=1}^B \nabla_{\theta} \ell(y_b, f(x_b; \theta))$$

The practical default. Typical batch sizes: 32–512.

Why mini-batches?

- **Computational efficiency:** Modern GPUs are optimised for batch operations. Processing 64 examples is often only marginally slower than processing 1.
- **Reduced variance:** Averaging over B examples reduces gradient variance by a factor of \sqrt{B} .
- **Regularisation effect:** The noise from using different mini-batches can help escape local minima and improve generalisation.

SGD Update Rule

$$\theta(t+1) = \theta(t) - \frac{\eta_t}{B} \sum_{b=1}^B \nabla_{\theta} \ell(y_b, f(x_b; \theta))$$

where:

- η_t = learning rate (possibly time-varying)
- B = mini-batch size
- $\nabla_{\theta} \ell$ = gradient of loss with respect to parameters

145.2 Learning Rate

NB!

The learning rate η is often the most important hyperparameter:

- **Too large:** Training diverges; loss oscillates or explodes
- **Too small:** Training is slow; may get stuck in poor local minima
- **Just right:** Rapid progress toward good solutions

Common strategies:

- Start with $\eta \in \{0.1, 0.01, 0.001\}$ and adjust based on loss curves
- Use learning rate schedules (decay over time)
- Use adaptive methods (Adam) that adjust rates automatically

Diagnosing learning rate problems:

- **Loss increases or oscillates wildly:** Learning rate too high. Try reducing by 3–10x.
- **Loss decreases very slowly:** Learning rate too low. Try increasing by 3–10x.
- **Loss decreases then plateaus:** May need learning rate decay, or the model has converged.

145.3 Optimisers

SGD with Momentum

Momentum accumulates gradients over time, smoothing updates:

$$v_t = \beta v_{t-1} + \nabla_\theta \mathcal{L}$$

$$\theta_t = \theta_{t-1} - \eta v_t$$

where $\beta \approx 0.9$ is the momentum coefficient.

Intuition: Momentum helps escape local minima and damps oscillations in narrow valleys.

Why momentum helps: Imagine rolling a ball down a loss landscape. Without momentum, the ball moves directly downhill at each point. With momentum, the ball builds up speed in consistent directions and can roll through small bumps.

In narrow valleys (where the loss is steep in one direction but shallow in another), vanilla SGD oscillates back and forth across the valley while making slow progress along it. Momentum accumulates the consistent “down the valley” component while averaging out the oscillating “across the valley” component.

Adam Optimiser

Adam combines momentum with adaptive learning rates:

Momentum (exponential average of gradients):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

Adaptive scaling (exponential average of squared gradients):

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Bias correction (important for early iterations):

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Update:

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Default values: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, $\eta = 0.001$

Understanding Adam:

- m_t : Like momentum, tracks the direction of recent gradients.
- v_t : Tracks the magnitude of recent gradients per parameter.
- **Division by $\sqrt{\hat{v}_t}$:** Parameters with large gradients get smaller effective learning rates; parameters with small gradients get larger effective rates. This is “adaptive.”
- **Bias correction:** At the start, m_t and v_t are biased toward zero (they are initialised to zero and slowly accumulate). The correction factors compensate for this.

Choosing an optimiser:

- **Adam**: Default for most deep learning; adaptive and robust
- **SGD with momentum**: Sometimes better final performance; requires more tuning
- **L-BFGS**: Small problems with smooth objectives; converges quickly

Visually

- **Input data**
- **Feature representations**
- **Outputs**
- **Loss function**
- **Optimizer**: 🤖

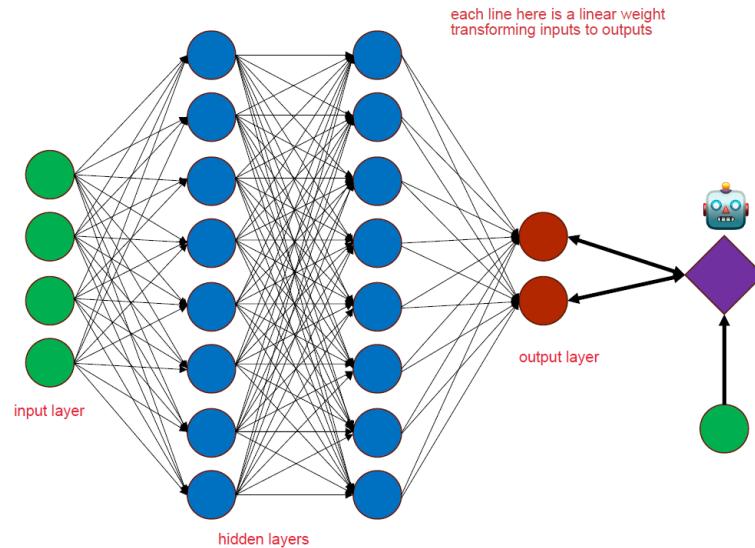


Figure 84: MLP design recipe: architecture, loss function, and optimiser.

MLP Design Choices

Architecture:

- Number of layers (depth)
- Units per layer (width)
- Activation functions

Loss function:

- Regression: MSE $\ell = (y - \hat{y})^2$
- Classification: Cross-entropy $\ell = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$

Optimiser:

- Adam (adaptive learning rates)—default choice
- SGD with momentum (often better final performance)
- L-BFGS (second-order; small problems only)

146 Initialisation

Section Summary

Proper weight initialisation is essential for training deep networks. Poor initialisation causes vanishing/exploding activations before training even begins. Xavier and He initialisations calibrate weight variance to maintain stable activation magnitudes across layers.

How we initialise the weights before training begins has a surprisingly large effect on whether training succeeds at all.

146.1 Why Initialisation Matters

The Problem with Naive Initialisation

Consider initialising all weights to the same value (e.g., zero or a constant):

- **All zeros:** All hidden neurons compute the same output. By symmetry, they receive the same gradient and remain identical forever. The network has effectively one neuron per layer.
- **All same constant:** Same problem—symmetry is not broken.
- **Too large:** Activations explode; sigmoid/tanh saturate immediately.
- **Too small:** Activations shrink toward zero through layers.

The symmetry problem: If all weights start identical, all neurons in a layer compute identical outputs. During backprop, they receive identical gradients. After the update, they remain identical. The network cannot learn different features in different neurons—it has collapsed to effectively one neuron per layer.

Solution: Random initialisation breaks symmetry. Each neuron starts slightly different, learns slightly different gradients, and evolves into a unique feature detector.

We need initialisations that:

1. Break symmetry (different neurons learn different features)
2. Maintain activation variance across layers
3. Maintain gradient variance across layers (for backprop)

146.2 Xavier/Glorot Initialisation

Xavier Initialisation

For a layer with n_{in} input neurons and n_{out} output neurons, initialise weights as:

$$W_{ij} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right)$$

or uniformly:

$$W_{ij} \sim \text{Uniform}\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right)$$

Derivation intuition: If $z = \sum_{i=1}^n w_i x_i$ and we want $\text{Var}(z) = \text{Var}(x)$, then with $\text{Var}(w_i) = \sigma_w^2$:

$$\text{Var}(z) = n \cdot \sigma_w^2 \cdot \text{Var}(x)$$

Setting $\sigma_w^2 = 1/n$ preserves variance in the forward pass. Xavier averages the requirements of forward and backward passes.

Deriving the variance: Assume inputs x_i are i.i.d. with mean 0 and variance $\text{Var}(x)$, and weights w_i are i.i.d. with mean 0 and variance σ_w^2 . Then:

$$\text{Var}(z) = \text{Var}\left(\sum_i w_i x_i\right) = \sum_i \text{Var}(w_i x_i) = \sum_i E[w_i^2]E[x_i^2] = n\sigma_w^2 \text{Var}(x)$$

To keep variance stable ($\text{Var}(z) = \text{Var}(x)$), we need $n\sigma_w^2 = 1$, so $\sigma_w^2 = 1/n$.

Xavier initialisation averages the forward and backward requirements: $\sigma_w^2 = 2/(n_{\text{in}} + n_{\text{out}})$.

Xavier initialisation was designed for sigmoid and tanh activations, which are roughly linear near zero.

146.3 He Initialisation

He Initialisation (for ReLU)

For ReLU activations, use:

$$W_{ij} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right)$$

Why different from Xavier? ReLU zeros out half the activations (those with negative pre-activations). This halves the variance, so we double the weight variance to compensate:

$$\text{Var}(z) = \frac{1}{2} \cdot n \cdot \sigma_w^2 \cdot \text{Var}(x)$$

Setting $\sigma_w^2 = 2/n$ maintains unit variance through ReLU.

Why ReLU needs different initialisation: After ReLU, roughly half the activations are zero (those with negative inputs). The variance of the output is therefore half the variance of the input. To compensate, we need twice the weight variance.

Initialisation Guidelines

- **Sigmoid/Tanh:** Xavier initialisation
- **ReLU/Leaky ReLU:** He initialisation
- **Biases:** Usually initialised to zero
- Modern frameworks (PyTorch, TensorFlow) use sensible defaults for common layer types

147 Vanishing and Exploding Gradients

Section Summary

In deep networks, gradients can vanish (approach zero) or explode (grow unboundedly) as they propagate through layers. This makes training unstable or ineffective. Solutions include careful initialisation, non-saturating activations (ReLU), gradient clipping, and normalisation techniques.

As networks become deeper, a fundamental problem emerges: gradients can become either extremely small or extremely large as they propagate backward through many layers.

Gradient Flow Through Depth

By the chain rule:

$$\frac{\partial \mathcal{L}}{\partial z^{(l)}} = \frac{\partial \mathcal{L}}{\partial z^{(L)}} \prod_{k=l}^{L-1} \frac{\partial z^{(k+1)}}{\partial z^{(k)}}$$

If derivatives between layers are approximately constant $\approx J$:

$$\frac{\partial \mathcal{L}}{\partial z^{(l)}} \approx J^{L-l} \frac{\partial \mathcal{L}}{\partial z^{(L)}}$$

Eigenvalue behaviour:

- $|\lambda_{\max}| < 1$: Gradients vanish exponentially with depth
- $|\lambda_{\max}| > 1$: Gradients explode exponentially with depth

The mathematics: Consider a simplified model where each layer multiplies by a constant factor λ . After $L - l$ layers, the gradient is scaled by λ^{L-l} .

- If $\lambda = 0.9$ and $L - l = 50$, the factor is $0.9^{50} \approx 0.005$ —gradients effectively vanish.
- If $\lambda = 1.1$ and $L - l = 50$, the factor is $1.1^{50} \approx 117$ —gradients explode.

Only when $\lambda \approx 1$ can we train deep networks reliably.

147.1 Solutions

147.1.1 Non-Saturating Activations (for vanishing gradients)

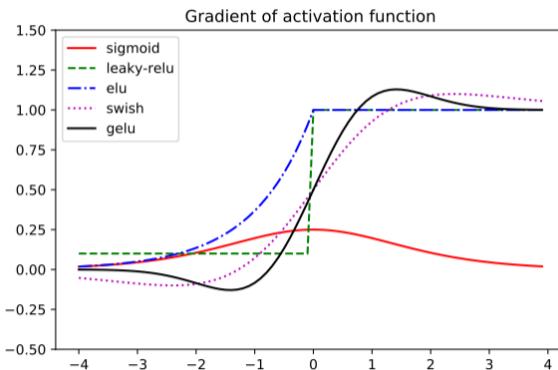


Figure 85: ReLU and Leaky ReLU maintain gradient flow better than sigmoid.

Activation Function Gradients

Sigmoid: Gradient $\rightarrow 0$ when $\sigma(z) \approx 0$ or $\sigma(z) \approx 1$ (saturation)

ReLU: Gradient is 1 for $z > 0$, 0 for $z < 0$ (dead neurons possible)

Leaky ReLU: Gradient is 1 for $z > 0$, α for $z < 0$ (no saturation, no dead neurons)

The sigmoid derivative $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ has maximum value 0.25 (at $z = 0$). Even in the best case, gradients shrink by 75% per layer. After 10 layers: $0.25^{10} \approx 10^{-6}$.

ReLU has derivative 1 for positive inputs, allowing gradients to flow unchanged. This was a key enabler of deep learning.

147.1.2 Gradient Clipping (for exploding gradients)

Gradient Clipping

$$g \leftarrow \min \left(1, \frac{c}{\|g\|} \right) g$$

Scales gradients to have maximum norm c , preserving direction while preventing instability.

How it works: If the gradient norm exceeds threshold c , we scale it down to exactly c . If the norm is below c , we leave it unchanged. The direction is always preserved.

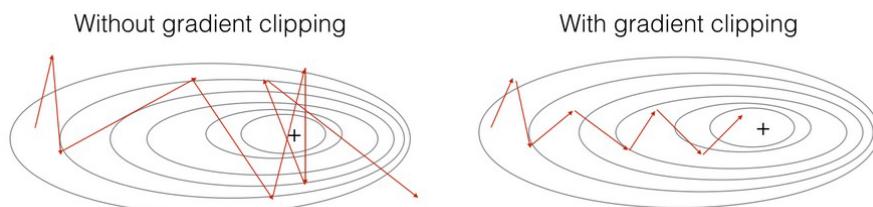


Figure 86: Gradient clipping constrains step size while preserving direction.

147.1.3 Batch Normalisation

Addresses **internal covariate shift**: the distribution of layer inputs changes during training as earlier layers update.

Batch Normalisation

$$y = \frac{x - \bar{x}_b}{\sqrt{\sigma_b^2 + \epsilon}} \cdot \gamma + \beta$$

where:

- \bar{x}_b, σ_b^2 = batch mean and variance
- γ, β = learned scale and shift parameters
- ϵ = small constant for numerical stability

At test time: Use exponential moving averages of training statistics.

What batch normalisation does: For each mini-batch, it normalises activations to have mean 0 and variance 1. Then it applies learnable scale (γ) and shift (β) parameters.

Why it helps:

- **Stable activations:** Keeps activations in a well-behaved range, preventing saturation.
- **Higher learning rates:** With normalised inputs, we can use larger learning rates.
- **Regularisation:** The noise from batch statistics provides implicit regularisation.

Benefits:

- Allows higher learning rates
- Provides regularisation effect
- Stabilises gradient flow

148 Regularisation

Section Summary

Neural networks with many parameters are prone to overfitting. Regularisation techniques constrain model complexity to improve generalisation. Weight decay penalises large weights; dropout forces redundant representations.

Neural networks often have more parameters than training examples. Without regularisation, they can memorise the training data perfectly while failing to generalise.

148.1 Weight Decay (L2 Regularisation)

Weight Decay

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \frac{\lambda}{2} \|\mathbf{w}\|^2$$

Penalises large weights, encouraging simpler models. Often implemented directly in the optimiser rather than as an explicit loss term.

Why weight decay works:

- **Prevents extreme weights:** Without regularisation, weights can grow arbitrarily large if it helps fit the training data, even slightly.
- **Smooth functions:** Large weights create sharp, wiggly decision boundaries. Small weights create smoother boundaries that generalise better.
- **Bayesian interpretation:** Weight decay corresponds to a Gaussian prior on weights—we prefer weights near zero a priori.

148.2 Dropout

Dropout

During training, randomly set a fraction p of activations to zero:

$$\tilde{h}_i = \frac{1}{1-p} \cdot h_i \cdot \text{Bernoulli}(1-p)$$

At test time, use all neurons without dropout (but scale by $1 - p$, or equivalently, scale training activations by $1/(1 - p)$).

Effect: Forces the network to learn redundant representations; acts as ensemble averaging over 2^n subnetworks.

Understanding dropout:

- During training, each forward pass uses a different random subset of neurons.
- The scaling factor $1/(1 - p)$ ensures the expected output is unchanged.
- At test time, we use all neurons, which approximates averaging over all possible dropout masks.

Why dropout works:

- **Prevents co-adaptation:** Neurons cannot rely on specific partners being present. Each neuron must be independently useful.
- **Ensemble effect:** Dropout trains an exponential number of “thinned” networks. The test-time network approximates the average of all these networks.
- **Redundant representations:** Forces the network to spread information across many neurons, making it robust to noise and missing features.

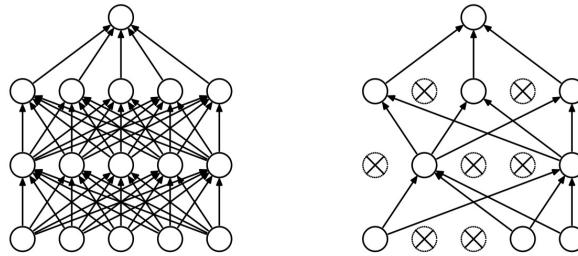


Figure 87: Dropout randomly removes connections during training.

Weight decay and dropout are complementary: weight decay constrains magnitudes; dropout encourages distributed representations.

149 Neural Networks as Gaussian Processes

Infinite-Width Limit

As the number of hidden units $\rightarrow \infty$, an MLP's output distribution converges to a Gaussian Process. The **Neural Tangent Kernel** (NTK) characterises this:

$$k(x, y) = \nabla_{\theta} f(x; \theta)^{\top} \nabla_{\theta} f(y; \theta)$$

In this limit, the NTK remains approximately constant during training, and the network behaves like kernel regression.

This is a remarkable theoretical connection that helps us understand neural networks through the lens of well-understood Gaussian processes.

The intuition: With randomly initialised weights, a neural network's output for any fixed input is a sum of many random terms (the contributions from each neuron). By the central limit theorem, this sum becomes Gaussian as the number of neurons grows. Moreover, the covariance between outputs at different inputs defines a kernel function.

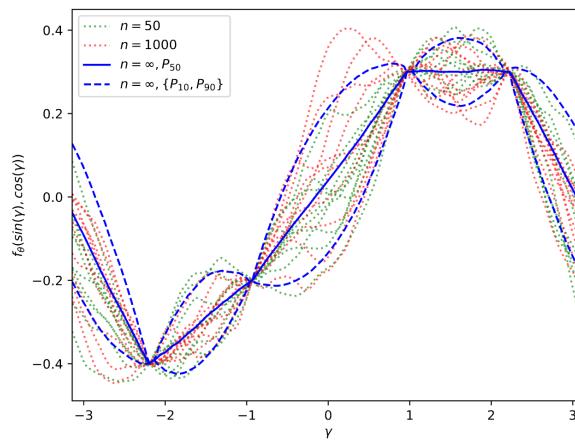


Figure 88: MLPs (red) as samples from a GP (blue) defined by the NTK.

The Neural Tangent Kernel: The NTK measures how similar the gradients are at two different inputs:

$$k(x, y) = \nabla_{\theta} f(x; \theta)^{\top} \nabla_{\theta} f(y; \theta)$$

If the gradients at x and y point in similar directions, these inputs are “similar” from the network’s perspective—updating the network to fit x will also affect its prediction at y .

This connection provides theoretical tools for understanding deep learning and bridges parametric (NNs) and non-parametric (GPs) approaches. It explains some phenomena like the ability of overparameterised networks to generalise despite having more parameters than data points.

Practical implications: While infinite-width limits are idealised, they suggest:

- Wider networks may be easier to analyse theoretically.
- Architecture choices (depth, activation functions) affect the implicit kernel.
- Overparameterised networks can generalise well, contrary to classical intuition.

150 Looking Ahead: Advanced Architectures

Connection to Week 11

The foundational concepts from this week—layers, activations, backpropagation, and training—extend to specialised architectures:

- **Convolutional Neural Networks (CNNs):** Exploit spatial structure in images through local connectivity and weight sharing. Convolution replaces full matrix multiplication, encoding the inductive bias that nearby pixels are related.
- **Recurrent Neural Networks (RNNs):** Process sequential data by maintaining hidden state across time steps. Backpropagation through time (BPTT) applies the same principles across the unrolled computation graph.
- **Attention Mechanisms:** Enable flexible, learned weighting of inputs. The Transformer architecture builds entirely on attention, achieving state-of-the-art results in NLP and increasingly in other domains.

Week 11 will develop these architectures in detail, showing how domain-specific inductive biases enable learning from images, sequences, and structured data.

151 Summary

Key Concepts from Week 10

1. **Biological inspiration:** Neurons inspire artificial networks, but the analogy is loose
2. **Perceptrons:** Linear classifiers with provable convergence for separable data
3. **Convergence theorem:** At most $(R/\gamma)^2$ updates for margin γ , radius R
4. **Linear separability:** Single-layer networks can only learn linearly separable functions
5. **XOR problem:** Demonstrates perceptron limitations; motivates multi-layer networks
6. **MLPs:** Learn feature representations through composed layers
7. **Non-linearity:** Essential—without it, depth provides no benefit
8. **Activation functions:** ReLU for hidden layers; sigmoid/softmax for outputs
9. **Loss functions:** MSE for regression; cross-entropy for classification
10. **Backpropagation:** Chain rule applied layer-by-layer; $O(W)$ complexity
11. **Training:** Mini-batch SGD; Adam as default optimiser
12. **Initialisation:** Xavier for sigmoid/tanh; He for ReLU
13. **Gradient problems:** Vanishing (use ReLU, batch norm); Exploding (use clipping)
14. **Regularisation:** Weight decay + dropout prevent overfitting
15. **Universal approximation:** One hidden layer suffices theoretically; depth helps empirically

Practical Checklist

When building an MLP:

1. **Architecture:** Start with 2–3 hidden layers, width 128–512
2. **Activation:** ReLU for hidden layers
3. **Output:** Linear (regression), sigmoid (binary), softmax (multi-class)
4. **Loss:** MSE (regression), cross-entropy (classification)
5. **Initialisation:** He for ReLU networks
6. **Optimiser:** Adam with default parameters
7. **Learning rate:** Start with 10^{-3} ; adjust based on training curves
8. **Regularisation:** Dropout (0.2–0.5) and/or weight decay (10^{-4})
9. **Batch size:** 32–256 typically
10. **Monitor:** Training and validation loss; watch for overfitting

152 Overview

Neural Networks II: Architecture Extensions

This week extends foundational neural network concepts to specialised architectures:

1. **Convolutional Neural Networks (CNNs)**: Exploit spatial structure in images
2. **Recurrent Neural Networks (RNNs)**: Maintain state for sequential data
3. **Attention mechanisms and Transformers**: Enable flexible, learnable weighting of inputs

The unifying theme is **composability**: building complex representations from simple, reusable components tailored to data structure.

The architectures we study this week are not arbitrary designs—they encode *inductive biases* that match the structure of different data types. CNNs encode the assumption that local patterns matter and should be detected regardless of position. RNNs encode the assumption that sequential order matters and past context should inform current predictions. Transformers encode the assumption that any element might be relevant to any other, letting the model learn which relationships matter. Choosing the right architecture is about matching these inductive biases to your data.

153 Neural Network Design Recap

Before diving into specialised architectures, we briefly review the core components that all neural networks share. These components—inputs, activations, connections, outputs, losses, and optimisers—combine in different ways to create the architectures we study this week.

153.1 Architecture Components

Network Design Choices

Inputs:

- Input layer dimensionality matches feature count
- Example: 28×28 image $\rightarrow 784$ input neurons (flattened)

Activation functions:

- ReLU: $f(x) = \max(0, x)$
- Sigmoid: $f(x) = \frac{1}{1+e^{-x}}$
- Tanh: $f(x) = \tanh(x)$
- Softmax (output layer for classification): $f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$

Connection patterns:

- Fully connected: Each neuron connected to all neurons in next layer
- Convolutional: Local receptive fields with shared weights
- Recurrent: Connections form loops for temporal dependencies

Output layer:

- Regression: Single neuron, linear activation
- Classification: K neurons with softmax for K classes

Unpacking the activation functions:

- **ReLU** is the most commonly used activation in hidden layers. It is computationally cheap (just a threshold) and does not saturate for positive inputs.
- **Sigmoid** squashes any real number to $(0, 1)$, historically used but now largely replaced by ReLU in hidden layers due to vanishing gradient issues. Still useful when you need a probability output.
- **Tanh** squashes to $(-1, 1)$, zero-centred which can help optimisation. Suffers from saturation at extremes like sigmoid.
- **Softmax** is used at the output layer for multi-class classification—it converts a vector of arbitrary real numbers into a probability distribution (all outputs positive and sum to 1).

153.2 Loss Functions

The loss function measures how well the network's predictions match the true labels. Different tasks require different loss functions.

Standard Loss Functions

Regression—Mean Squared Error:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Unpacking: For each example i , we compute the squared difference between the true value y_i and prediction \hat{y}_i . The average over all n examples gives the loss. Squaring penalises large errors more than small ones and makes the loss differentiable everywhere.

Binary Classification—Cross-Entropy (Log Loss):

$$\mathcal{L}_{\text{CE}} = -\frac{1}{n} \sum_{i=1}^n [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

Unpacking: Here $y_i \in \{0, 1\}$ is the true class and $\hat{y}_i \in (0, 1)$ is the predicted probability of class 1. When $y_i = 1$, only the $\log(\hat{y}_i)$ term contributes—we want \hat{y}_i close to 1. When $y_i = 0$, only the $\log(1 - \hat{y}_i)$ term contributes—we want \hat{y}_i close to 0. The negative sign ensures the loss is positive and minimised when predictions are confident and correct.

Multi-class Classification—Categorical Cross-Entropy:

$$\mathcal{L}_{\text{CCE}} = -\frac{1}{n} \sum_{i=1}^n \sum_{c=1}^C y_{i,c} \log(\hat{y}_{i,c})$$

where $y_{i,c}$ is the one-hot encoded true label and $\hat{y}_{i,c}$ is the predicted probability for class c .

Unpacking: The one-hot encoding means $y_{i,c} = 1$ for exactly one class and 0 for all others. So this sum collapses to $-\log(\hat{y}_{i,c^*})$ where c^* is the true class—we simply want high predicted probability for the correct class.

153.3 Optimisers

Neural networks are trained by gradient descent: iteratively adjusting parameters to reduce the loss. Different optimisers use different strategies for determining the direction and magnitude of parameter updates.

Gradient-Based Optimisation

Stochastic Gradient Descent (SGD):

$$w_{t+1} = w_t - \eta \nabla \mathcal{L}(w_t)$$

Unpacking: At each step t , we compute the gradient of the loss with respect to the current weights, $\nabla \mathcal{L}(w_t)$, and take a step of size η (the learning rate) in the negative gradient direction. “Stochastic” refers to computing gradients on mini-batches rather than the full dataset.

Adam combines momentum with adaptive learning rates:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \\ \hat{m}_t &= \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \\ w_{t+1} &= w_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \end{aligned}$$

Unpacking each line:

1. m_t is an exponential moving average of gradients (momentum)—helps smooth out noisy gradients and accelerate in consistent directions
2. v_t is an exponential moving average of squared gradients—estimates the variance of gradients for each parameter
3. \hat{m}_t and \hat{v}_t are bias-corrected estimates (early in training, the moving averages are biased towards zero)
4. The update divides by $\sqrt{\hat{v}_t}$ to give each parameter its own adaptive learning rate—parameters with large gradients get smaller updates

Adam is generally the default choice for deep learning.

Other optimisers: AdaGrad and RMSProp (which Adam builds upon), SGD with momentum, and second-order methods like BFGS (expensive but can converge faster for smaller models).

Interactive exploration: TensorFlow Playground (playground.tensorflow.org) provides an excellent interactive environment for building intuition about how network architecture, activation functions, and optimisation interact.

154 Vanishing and Exploding Gradients

Gradient-based training propagates error signals backward through layers via the chain rule. In deep networks, this can become unstable. Understanding this problem is essential for designing and training deep architectures.

154.1 The Problem

Gradient Flow in Deep Networks

For a network with L layers, backpropagation computes:

$$\frac{\partial \mathcal{L}}{\partial z_l} = \frac{\partial \mathcal{L}}{\partial z_L} \cdot \prod_{k=l}^{L-1} \frac{\partial z_{k+1}}{\partial z_k}$$

where z_k denotes the pre-activation at layer k .

If layer-wise derivatives are approximately constant $\frac{\partial z_{k+1}}{\partial z_k} \approx J$, then:

$$\frac{\partial \mathcal{L}}{\partial z_l} \approx J^{L-l} \cdot \frac{\partial \mathcal{L}}{\partial z_L}$$

Unpacking this formula: The gradient at layer l (which determines how we update that layer's weights) depends on a *product* of $L - l$ layer-wise derivatives. If we have 100 layers and we're computing the gradient for layer 1, we're multiplying together 99 terms. Even if each term is close to 1, this product can become very large or very small.

The eigenvalue criterion: For a matrix J representing the layer-wise Jacobian, the behaviour depends on its eigenvalues λ :

- $|\lambda| < 1$: $\lim_{L \rightarrow \infty} J^{L-l} = 0$ (vanishing gradients)
- $|\lambda| > 1$: $\lim_{L \rightarrow \infty} J^{L-l} = \infty$ (exploding gradients)

Analogy: Imagine playing a game of “telephone” where each person multiplies the message by some factor before passing it on. If each person multiplies by 0.9, after 100 people the message has been reduced to $0.9^{100} \approx 0$ (vanished). If each multiplies by 1.1, after 100 people it's $1.1^{100} \approx 10^4$ (exploded).

Practical consequences:

- **Vanishing gradients:** Early layers receive negligible updates; the network cannot learn useful features in lower layers, effectively wasting those parameters
- **Exploding gradients:** Weight updates become erratic; training diverges with NaN losses or wildly oscillating parameters

154.2 Gradient Clipping

Gradient Clipping

When gradients exceed a threshold c , scale them to prevent instability:

$$g' = \min\left(1, \frac{c}{\|g\|}\right) \cdot g$$

This preserves gradient **direction** while constraining **magnitude**.

Unpacking the formula:

- $\|g\|$ is the norm (magnitude) of the gradient vector
- If $\|g\| \leq c$, the factor $\min(1, c/\|g\|) = 1$, so $g' = g$ (no change)

- If $\|g\| > c$, the factor is $c/\|g\| < 1$, so $g' = (c/\|g\|) \cdot g$ has magnitude exactly c
- The direction of g is preserved in both cases—we're just limiting how far we step

Interpretation as adaptive learning rate: Gradient clipping can be viewed as dynamically reducing the effective learning rate when gradients are large:

$$g' = \frac{c}{\|g\|} \cdot g \implies \text{effective learning rate} = \eta \cdot \frac{c}{\|g\|}$$

Large gradients automatically trigger smaller steps, stabilising training without requiring manual learning rate tuning for each situation.

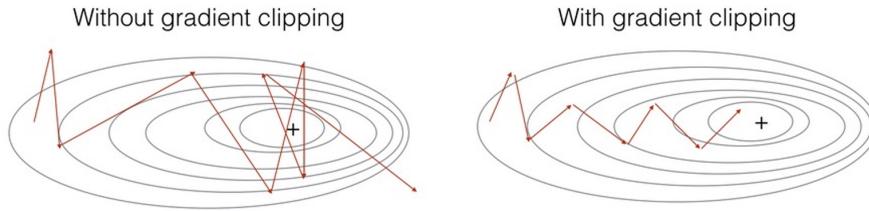


Figure 89: Gradient clipping constrains update magnitude while preserving direction, preventing divergence in optimisation. Without clipping (red), large gradients cause the optimisation to overshoot and diverge. With clipping (blue), the direction is maintained but the step size is bounded.

154.3 Vanishing Gradients and Activation Functions

The choice of activation function critically affects gradient flow. The issue is *saturation*: when activations are in regions where the derivative is near zero, gradients cannot flow.

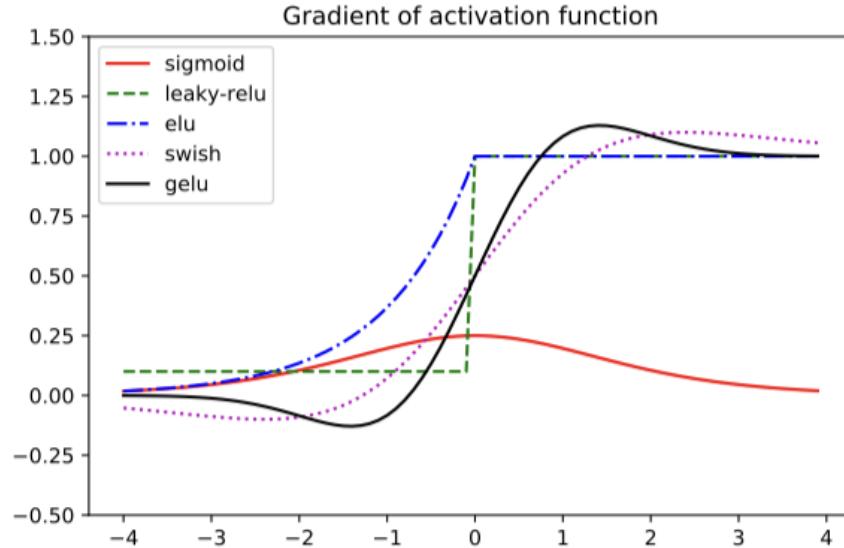


Figure 90: Activation functions and their derivatives. Sigmoid saturates for large inputs, causing vanishing gradients. ReLU maintains gradient of 1 for positive inputs.

Activation Function Gradients

Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$

$$\frac{\partial \mathcal{L}}{\partial x} \propto \sigma(x)(1 - \sigma(x))$$

Gradient $\rightarrow 0$ when $\sigma(x) \rightarrow 0$ or $\sigma(x) \rightarrow 1$ (saturation).

Unpacking: The sigmoid derivative $\sigma(x)(1 - \sigma(x))$ has maximum value 0.25 (when $\sigma(x) = 0.5$). For inputs $|x| > 4$, the derivative is essentially zero. If any layer produces large pre-activations, gradients die at that layer.

ReLU: $\text{ReLU}(x) = \max(0, x)$

$$\frac{\partial \text{ReLU}}{\partial x} = \mathbb{I}(x > 0)$$

Gradient is 1 for positive inputs (no saturation), but exactly 0 for negative inputs.

Unpacking: ReLU solves the saturation problem for $x > 0$ —the gradient is always 1, no matter how large the input. But for $x \leq 0$, the gradient is exactly 0. This creates “dead neurons”: if a neuron’s inputs consistently produce negative pre-activations, it never updates and contributes nothing to learning.

Leaky ReLU: $\text{LeakyReLU}(x) = \max(\alpha x, x)$

$$\frac{\partial \text{LeakyReLU}}{\partial x} = \begin{cases} 1 & x > 0 \\ \alpha & x \leq 0 \end{cases}$$

Non-zero gradient everywhere (typically $\alpha = 0.01$).

Unpacking: Leaky ReLU addresses dead neurons by having a small positive slope (α) for negative inputs. The gradient is always non-zero, so neurons can always update. The small α means negative inputs are still “suppressed” but not completely zeroed.

Activation Function Summary

- **Sigmoid:** Vanishes for large $|x|$; avoid in hidden layers of deep networks
- **ReLU:** No saturation for $x > 0$; risk of “dead neurons” when $x < 0$
- **Leaky ReLU:** Best of both worlds—non-saturating and no dead neurons

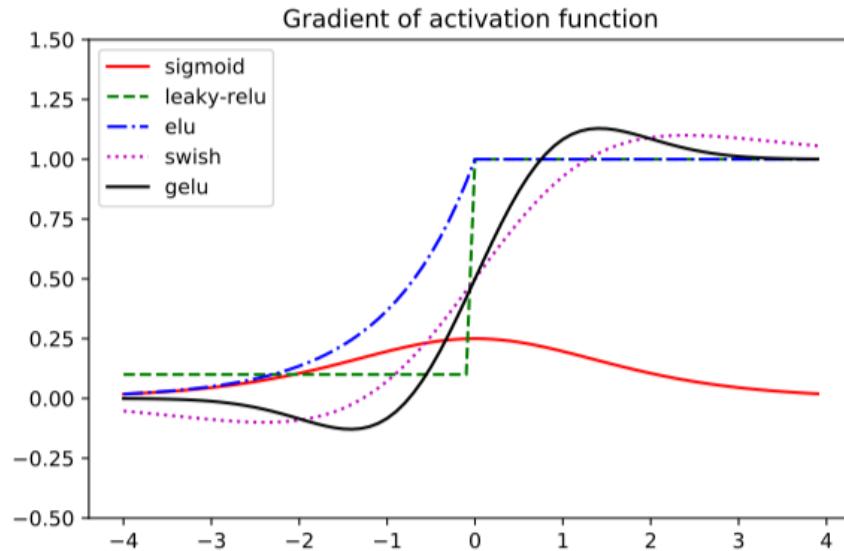


Figure 91: Non-saturating activation functions like ReLU and its variants maintain gradient flow, enabling training of deeper networks.

154.4 Batch Normalisation

Batch normalisation stabilises training by normalising layer activations, preventing them from drifting to saturating regions.

Batch Normalisation

$$y = \frac{x - \bar{x}_{\text{batch}}}{\sqrt{\sigma_{\text{batch}}^2 + \epsilon}} \cdot \gamma + \beta$$

where:

- $\bar{x}_{\text{batch}}, \sigma_{\text{batch}}^2$ = batch mean and variance
- γ, β = learned scale and shift parameters
- ϵ = small constant for numerical stability

At test time: Use exponential moving averages of training statistics.

Unpacking the three steps:

1. **Normalise:** Subtract batch mean and divide by batch standard deviation. This centres activations at zero with unit variance, preventing drift to saturation regions.
2. **Scale and shift:** The learnable parameters γ and β allow the network to “undo” the normalisation if beneficial. If $\gamma = \sigma_{\text{batch}}$ and $\beta = \bar{x}_{\text{batch}}$, we recover the original distribution.
3. **Test time:** We cannot compute batch statistics from a single example. Instead, we use exponential moving averages accumulated during training, which estimate the population statistics.

Why this helps:

- **Reduces internal covariate shift:** As earlier layers update, the distribution of inputs to later layers changes. Batch normalisation keeps these distributions stable.
- **Allows higher learning rates:** More stable gradients permit more aggressive optimisation without divergence.
- **Acts as implicit regularisation:** The noise from batch statistics (which vary between mini-batches) adds a regularising effect similar to dropout.
- **Prevents saturation:** Normalised activations stay in the “active” region of activation functions.

NB!

Training vs inference mode: Batch normalisation behaves differently at training time (uses batch statistics) versus inference time (uses accumulated statistics). Always ensure your model is in the correct mode (`model.train()` vs `model.eval()`) in PyTorch). Using training-mode batch norm at inference produces inconsistent, unreliable results.

154.5 Regularisation

Regularisation techniques prevent overfitting by constraining model complexity. Two primary methods are used in neural networks.

154.5.1 Weight Decay (L2 Regularisation)

Weight Decay

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \frac{\lambda}{2} \|w\|^2$$

Penalises large weights, encouraging simpler models. Equivalent to ridge regression applied to all network parameters.

Unpacking:

- $\|w\|^2 = \sum_i w_i^2$ is the sum of squared weights
- λ controls regularisation strength—larger λ penalises large weights more
- The factor of $\frac{1}{2}$ is conventional, giving a clean gradient: $\frac{\partial}{\partial w} \frac{\lambda}{2} \|w\|^2 = \lambda w$
- This gradient term effectively “shrinks” weights towards zero at each update

Intuition: Large weights allow the network to fit complex, potentially spurious patterns in the training data. Penalising weight magnitude encourages smoother functions that generalise better. Networks with smaller weights tend to be more “conservative” in their predictions.

Weight decay is typically implemented directly in the optimiser (the `weight_decay` parameter) rather than added to the loss function, as this is more computationally efficient.

154.5.2 Dropout

Dropout

During training, randomly set each neuron's output to zero with probability p :

$$\tilde{h}_i = \frac{1}{1-p} \cdot h_i \cdot \text{Bernoulli}(1-p)$$

At test time, use all neurons (no dropout).

Unpacking the formula:

- Bernoulli($1-p$) is a random variable that is 1 with probability $1-p$ and 0 with probability p
- When the Bernoulli is 0, the neuron's output is zeroed (dropped)
- The factor $\frac{1}{1-p}$ scales up surviving neurons to maintain the expected output magnitude
- At test time, all neurons are used without dropout or scaling (the training scaling ensures expectations match)

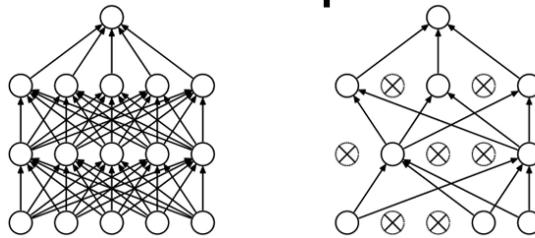


Figure 92: Dropout randomly removes neurons during training, forcing redundant representations. Each forward pass uses a different random subset of neurons.

Why dropout works:

1. **Prevents co-adaptation:** Without dropout, neurons can learn to rely on specific other neurons being present. Dropout forces each neuron to be useful independently.
2. **Ensemble interpretation:** Dropout can be viewed as implicitly training an ensemble of 2^n subnetworks (one for each dropout mask), then averaging their predictions at test time.
3. **Uncertainty estimation:** Applying dropout at test time and averaging multiple forward passes provides a form of uncertainty quantification—predictions that vary significantly under different masks indicate high model uncertainty.

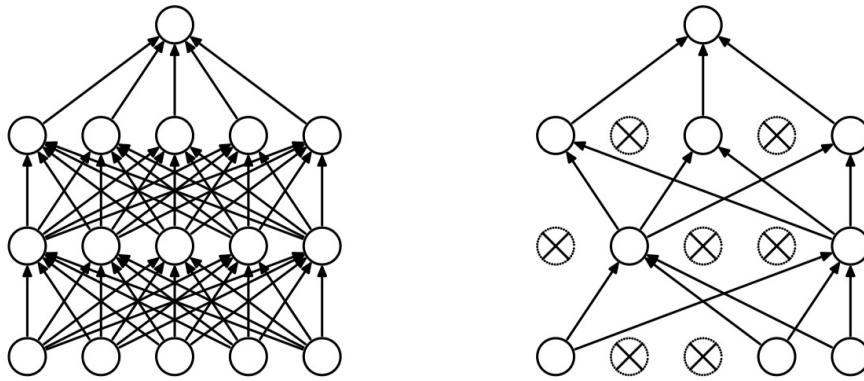


Figure 93: Effect of regularisation: without regularisation (left), the model overfits to training noise. With regularisation (right), the model learns smoother decision boundaries that generalise better.

155 Convolutional Neural Networks

CNNs: Key Idea

CNNs exploit **spatial structure** by:

- Using local receptive fields (each neuron sees a small region)
- Sharing weights across spatial positions (translation equivariance)
- Building hierarchies from local features to global representations

Result: Massive parameter reduction compared to fully connected networks, with built-in inductive bias for spatial data.

155.1 Motivation: Why Not Fully Connected?

Images present unique challenges that standard neural networks handle poorly. Understanding these challenges motivates the convolutional architecture.

High dimensionality: A 256×256 RGB image has $256 \times 256 \times 3 = 196,608$ features—and they are continuous, not categorical.

Parameter explosion: A fully connected layer from this input to just 1000 hidden units requires $196,608 \times 1000 \approx 200$ million parameters—for a single layer. This is both computationally prohibitive and prone to severe overfitting with any reasonable training set size.

Local structure matters: Unlike tabular data where feature order is often arbitrary (we could shuffle columns without changing the problem), images have meaningful spatial relationships. Neighbouring pixels are correlated; edges and textures emerge from local patterns. Fully connected layers ignore this structure entirely—they treat a pixel at position $(0,0)$ and one at $(255, 255)$ as equally related.

Translation invariance: A face should be recognised regardless of its position in the image. Fully connected networks treat each pixel position independently, missing this structure—the network would need to learn separate detectors for a face at every possible location, requiring exponentially more parameters and data.

Continuous features: Pixel intensities are continuous (0–255 or 0–1), unlike discrete categories in much tabular data. This continuity requires smooth transformations, not arbitrary mappings.

155.2 The Convolution Operation

The key insight: instead of processing the entire image at once, slide a small filter (kernel) across the image to extract local features.

Convolution Definition

Continuous (1D):

$$(f * g)(z) = \int_{\mathbb{R}} f(u) g(z - u) du$$

Unpacking: This is the mathematical definition of convolution from signal processing. We “flip” function g , slide it along f , and at each position compute the integral of their product. The output at position z depends on the weighted overlap of f and g around z .

Discrete 1D (filter w with k elements applied to input x):

$$(w * x)_i = \sum_{j=1}^k w_j \cdot x_{i+j-1}$$

Unpacking: For each output position i , we take k consecutive input elements starting at position i , multiply each by the corresponding filter weight w_j , and sum. The filter “slides” along the input, computing this weighted sum at each position.

Discrete 2D (filter W of size $k \times k$ applied to input X):

$$(W * X)_{i,j} = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} W_{m,n} \cdot X_{i+m, j+n}$$

Unpacking: Same idea in two dimensions. For each output position (i, j) , we take a $k \times k$ patch of the input centred around that position, multiply element-wise by the filter W , and sum. The filter slides across both dimensions of the image.

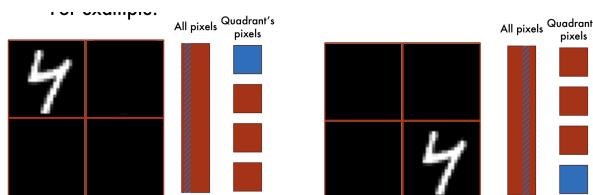


Figure 94: A convolution filter sliding across an image, computing local weighted sums. The filter “looks at” one small region at a time, extracting local features.

Why overlapping windows matter:

What if we simply divided the image into non-overlapping regions and processed each separately? Objects that span region boundaries would be split, making them hard to recognise.

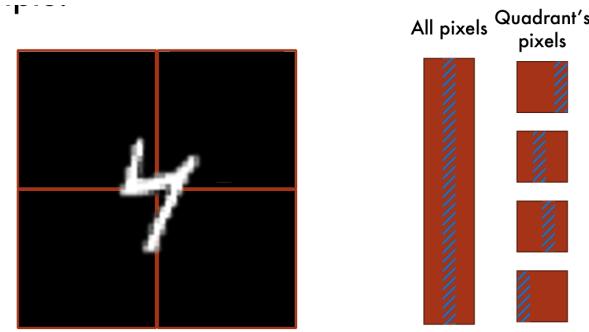


Figure 95: When objects span multiple regions, convolutions capture features that simple partitioning would miss. The sliding window ensures every local pattern is detected regardless of position.

Convolutions solve this by using overlapping windows. By sliding the filter one pixel at a time, we ensure that every local pattern is captured, regardless of where it appears in the image.

$$\begin{array}{c} \text{Input} \quad \quad \quad \text{Kernel} \quad \quad \quad \text{Output} \\ \boxed{0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6} \quad * \quad \boxed{1 \ 2} \quad = \quad \boxed{2 \ 5 \ 8 \ 11 \ 14 \ 17} \end{array}$$

Figure 96: 1D convolution: the filter weights determine which local patterns are detected. Different weights detect different patterns (edges, gradients, flat regions).

$$[W * X](i, j) = \sum_{u=0}^{H-1} \sum_{v=0}^{W-1} w_{u,v} x_{i+u, j+v}$$

Figure 97: 2D convolution for image processing, extending the same principle to spatial data.

Learned filters: The network **learns** the filter weights during training—we do not hand-design them. Different filters detect different features: edges, textures, shapes. The learning algorithm discovers which local patterns are useful for the task.

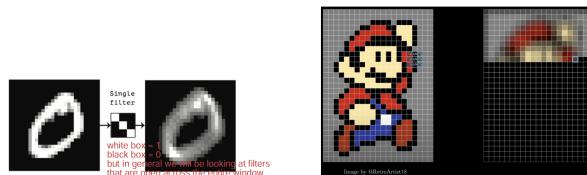


Figure 98: Learned filters detect interpretable features such as horizontal and vertical edges. Early layers typically learn edge detectors; deeper layers learn more complex patterns.

155.3 Feature Maps and Channels

Real images have multiple channels (e.g., RGB). A convolutional layer handles this by extending filters to 3D.

Multi-Channel Convolution

Real images have multiple channels (e.g., RGB). A convolutional layer handles this by:

Input: $X \in \mathbb{R}^{H \times W \times C_{\text{in}}}$ (height \times width \times input channels)

Filter: $W \in \mathbb{R}^{k \times k \times C_{\text{in}}}$ (one 3D filter spans all input channels)

Output (one feature map):

$$Y_{i,j} = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} \sum_{c=1}^{C_{\text{in}}} W_{m,n,c} \cdot X_{i+m,j+n,c} + b$$

Multiple filters: Apply C_{out} different filters to produce C_{out} feature maps (output channels).

Unpacking:

- For an RGB image, $C_{\text{in}} = 3$. The filter has weights for each colour channel at each spatial position.
- The sum over c combines information from all input channels into a single output value.
- Each filter produces one 2D feature map. Using 64 filters produces 64 feature maps, each potentially detecting a different type of pattern.
- The bias b is typically shared across all spatial positions for a given filter.

Each feature map captures a different type of pattern—one might detect horizontal edges, another vertical edges, another corners. The collection of feature maps forms a rich representation of the input.

155.4 Convolution as Sparse Matrix Multiplication

Understanding convolution as matrix multiplication provides insight into its computational structure.

Matrix View of Convolution

Convolution can be expressed as matrix multiplication $y = Cx$ where C is a sparse, structured matrix:

$$C = \begin{bmatrix} w_1 & w_2 & 0 & 0 & \dots \\ 0 & w_1 & w_2 & 0 & \dots \\ 0 & 0 & w_1 & w_2 & \dots \\ \vdots & & & & \ddots \end{bmatrix}$$

The same weights appear repeatedly (weight sharing), and most entries are zero (local connectivity).

Unpacking the structure:

- Each row computes one output element
- The non-zero entries are the filter weights (w_1, w_2, \dots)
- Each row shifts the pattern by one position, implementing the “sliding” of the filter

- Zeros mean those input positions do not affect that output—locality

Matrix multiplication.

$$y = Cx = \left(\begin{array}{ccc|ccc|ccc} w_1 & w_2 & 0 & w_3 & w_4 & 0 & 0 & 0 & 0 \\ 0 & w_1 & w_2 & 0 & w_3 & w_4 & 0 & 0 & 0 \\ 0 & 0 & 0 & w_1 & w_2 & 0 & w_3 & w_4 & 0 \\ 0 & 0 & 0 & 0 & w_1 & w_2 & 0 & w_3 & w_4 \end{array} \right) \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix}$$

Figure 99: Convolution as sparse matrix multiplication: zeros correspond to pixels outside the receptive field.

This structure dramatically reduces parameters compared to fully connected layers and encodes the inductive bias that local patterns matter more than distant correlations.

15.5 Parameter Sharing and Translation Equivariance

Parameter Sharing Benefits

Parameter count comparison for a $224 \times 224 \times 3$ input:

Fully connected to 1000 units:

$$(224 \times 224 \times 3) \times 1000 = 150,528,000 \text{ parameters}$$

Convolutional layer with 64 filters of size 3×3 :

$$(3 \times 3 \times 3) \times 64 + 64 = 1,792 \text{ parameters}$$

This is a reduction by a factor of nearly 100,000.

Translation equivariance: Because the same filter is applied at every position, if the input shifts, the output shifts by the same amount. Formally, if T is a translation operator:

$$\text{Conv}(T[X]) = T[\text{Conv}(X)]$$

Unpacking: If we shift an edge 10 pixels to the right in the input image, the edge detector's response shifts 10 pixels to the right in the output. The network does not need to learn separate edge detectors for every possible location—one edge detector works everywhere. This is a fundamental inductive bias of CNNs.

155.6 Receptive Field

Receptive Field

The **receptive field** of a neuron is the region of the input that influences its activation.

For a single convolutional layer with kernel size k : receptive field = $k \times k$.

For stacked layers, the receptive field grows:

$$r_l = r_{l-1} + (k_l - 1) \times \prod_{i=1}^{l-1} s_i$$

where r_l is the receptive field at layer l , k_l is the kernel size, and s_i are the strides of preceding layers.

Unpacking: Each additional layer expands the receptive field. The product of strides accounts for how earlier layers' downsampling affects the mapping from input to later layers.

Deeper networks have larger receptive fields, allowing later layers to capture more global patterns. This is why CNNs build from local features (edges) to global features (objects).

Example: Three stacked 3×3 convolutions have a 7×7 receptive field, but use fewer parameters than a single 7×7 convolution and apply more nonlinearities (one after each layer), adding representational power.

155.7 Padding and Strides

155.7.1 Padding

Padding Types

Valid convolution (no padding): Output shrinks as filters cannot be centred on edge pixels.

$$\text{Output size} = \text{Input size} - k + 1$$

Same convolution (zero padding): Pad with $\lfloor k/2 \rfloor$ zeros on each side so output dimensions match input dimensions.

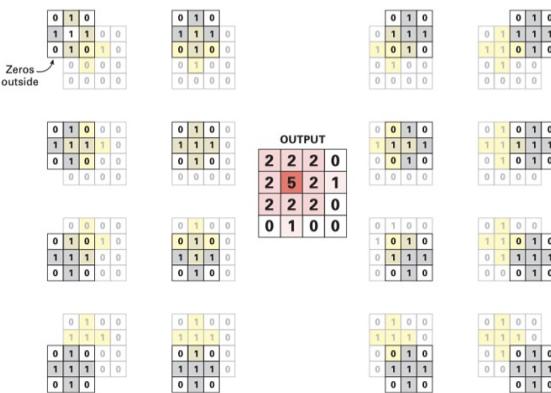


Figure 100: Zero-padding allows the filter to process edge regions, preserving spatial dimensions.

Why padding matters: Without padding, each convolutional layer shrinks the spatial dimensions. After many layers, the feature maps become tiny, losing spatial information. Same-padding

maintains dimensions, allowing very deep networks while preserving spatial resolution.

155.7.2 Strides

Adjacent filter positions produce highly correlated outputs (they share most of their inputs). Strides skip positions to reduce redundancy and spatial dimensions.

Stride Effect on Output Size

For input size n , kernel size k , padding p , and stride s :

$$\text{Output size} = \left\lceil \frac{n + 2p - k}{s} \right\rceil + 1$$

- **Stride 1:** Filter moves one pixel per step (default)
- **Stride 2:** Filter skips one pixel; output spatial dimensions approximately halved
- **Stride s :** Output dimensions reduced by factor of approximately s

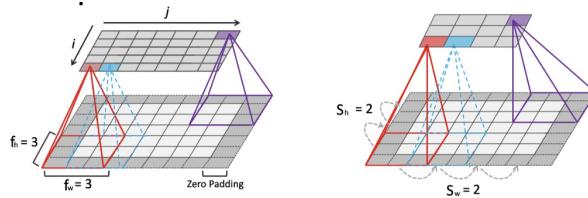


Figure 101: Comparison of stride 1 (left) vs stride 2 (right). Larger strides downsample the feature map.

Intuition: With stride 1 and a 3×3 filter, adjacent output positions share about 66% of their receptive field. This redundancy is computationally expensive and often unnecessary. Larger strides produce more compact representations with reduced spatial resolution.

155.8 Pooling

Pooling provides translation invariance by summarising local regions.

Pooling Operations

For a 2×2 pooling region with stride 2:

- **Max pooling:** Output the maximum value in the region
- **Average pooling:** Output the mean value in the region
- **Global average pooling:** Average over the entire spatial extent (used before final classifier)

Max pooling is more common in hidden layers; it selects the strongest activation in each region.

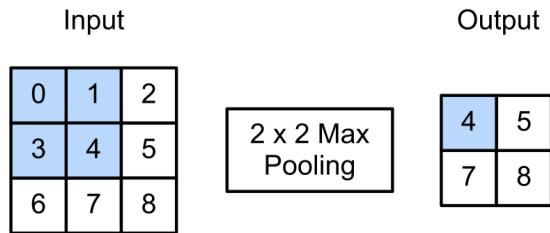


Figure 102: Max pooling selects the largest value in each region, providing translation robustness.

Translation invariance vs equivariance:

- **Convolution** is translation *equivariant*: shifting input shifts output proportionally
- **Pooling** is approximately translation *invariant*: small input shifts may not change output

Pooling deliberately discards exact spatial information, helping the network care about *what* features are present rather than *exactly where* they are.

NB!

During backpropagation with max pooling, gradients flow only through the position of the maximum value. Other positions receive zero gradient—they did not contribute to the output. This sparse gradient flow can slow learning but also acts as a form of regularisation.

Pooling vs strided convolutions: Modern architectures often use strided convolutions instead of pooling, as they are learnable and can capture more nuanced downsampling strategies.

155.9 CNN Architecture

A typical CNN alternates convolutional and pooling layers, building a hierarchy of increasingly abstract features.

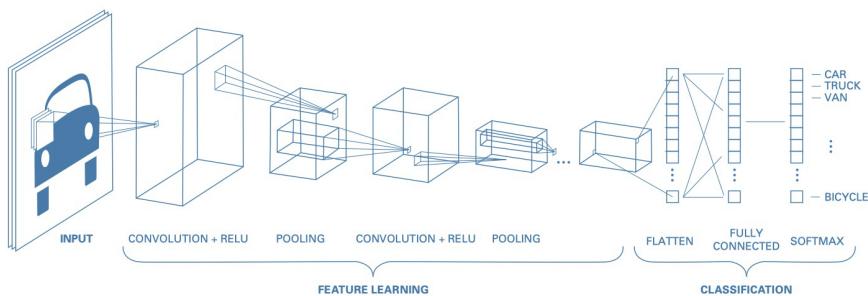


Figure 103: CNN architecture: early layers detect edges, middle layers detect parts, deep layers detect objects.

Standard CNN Structure

1. **Convolutional layers:** Extract local features with learned filters
2. **Activation:** Apply ReLU (or variant) after each convolution
3. **Pooling layers:** Downsample and provide translation invariance
4. **Repeat:** Stack conv/pool blocks to build hierarchical representations
5. **Flatten:** Convert final feature maps to a vector
6. **Fully connected layers:** Map flattened features to outputs
7. **Output:** Softmax for classification, linear for regression

Hierarchical feature learning:

- Early layers: Low-level features (edges, textures, colours)
- Middle layers: Mid-level features (parts, shapes, patterns)
- Deep layers: High-level features (objects, scenes, concepts)

This progression from local to global, simple to complex, mirrors how biological visual systems work and is key to CNNs' success. Each layer builds on the features detected by previous layers.

155.10 Classic CNN Architectures

Understanding the evolution of CNN architectures reveals key insights about what makes deep networks work.

Evolution of CNN Architectures

LeNet-5 (1998): First successful CNN for digit recognition.

- 2 convolutional + 2 pooling + 3 FC layers
- $\sim 60K$ parameters
- Introduced the conv-pool-conv-pool-FC pattern

AlexNet (2012): Won ImageNet, sparked the deep learning revolution.

- 5 conv + 3 FC layers, $\sim 60M$ parameters
- Key innovations: ReLU activation, dropout, data augmentation, GPU training
- Demonstrated that deeper networks with more data dramatically outperform classical methods

VGGNet (2014): Showed that depth matters.

- 16–19 layers using only 3×3 convolutions
- $\sim 138M$ parameters
- Insight: stacking small filters is better than using large filters (more nonlinearities, fewer parameters)

ResNet (2015): Enabled training of very deep networks (50–152+ layers).

- Introduced residual (skip) connections
- Won ImageNet with 3.6% error (better than human performance)
- Key insight: learning residuals is easier than learning direct mappings

CNN Architecture Evolution

- **Trend 1:** Deeper networks generally perform better (with appropriate training techniques)
- **Trend 2:** Smaller filters (3×3) stacked are more effective than large filters
- **Trend 3:** Skip connections are essential for training very deep networks
- **Trend 4:** Global average pooling replacing large FC layers reduces overfitting

155.11 Residual Connections

The key innovation enabling very deep networks.

Residual (Skip) Connections

Instead of learning a direct mapping $H(x)$, learn the **residual** $F(x) = H(x) - x$:

$$y = F(x) + x$$

A **residual block** computes:

$$y = \sigma(W_2 \cdot \sigma(W_1 x + b_1) + b_2) + x$$

where σ is the activation function.

If dimensions do not match, use a linear projection:

$$y = F(x) + W_s x$$

Unpacking: The “ $+x$ ” is the skip connection—it adds the input directly to the output of the convolutional layers. The layers only need to learn $F(x) = H(x) - x$, the difference between desired output and input.

Why residual connections work:

1. **Identity mapping baseline:** If the optimal transformation is close to identity, the network only needs to learn $F(x) \approx 0$, which is easier than learning $H(x) \approx x$ from scratch. Setting all weights to zero gives identity—a much better initialisation than random.
2. **Gradient flow:** During backpropagation, the skip connection provides a direct path for gradients:

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \left(\frac{\partial F}{\partial x} + 1 \right)$$

The “ $+1$ ” term ensures gradients flow even if $\frac{\partial F}{\partial x}$ is small, mitigating vanishing gradients.

3. **Ensemble interpretation:** A ResNet can be viewed as an implicit ensemble of shallower networks—different paths through the skip connections correspond to different effective depths.

Residual Connections Summary

- Enable training of networks with 100+ layers
- Provide shortcut paths for gradient flow
- Make it easy for layers to learn identity mappings when needed
- Now standard in almost all deep architectures

When CNNs Work Well

CNNs excel when:

- **Local structure matters:** Neighbouring elements are correlated
- **Translation invariance is desired:** Pattern location is less important than pattern presence
- **Hierarchy exists:** Complex patterns compose from simpler ones

Examples: images, spectrograms, genomic sequences, medical imaging.

NB!

CNNs struggle with data requiring **long-range dependencies**. In text or audio, a word early in a sequence may relate to a word much later—CNNs' local receptive fields miss these connections unless the network is very deep. Use RNNs, LSTMs, or Transformers for such tasks.

156 Recurrent Neural Networks

RNNs: Key Idea

RNNs maintain **hidden state** that evolves over time, enabling:

- Variable-length input/output sequences
- Implicit dependence on all previous inputs
- Memory of past context

Core equation: $h_t = f(h_{t-1}, x_t; \theta)$ — the same function applied at every timestep.

Unlike feedforward networks that process fixed-size inputs, RNNs process sequences of arbitrary length by maintaining a “memory” (the hidden state) that summarises everything seen so far.

156.1 Architecture

RNN Formulation

At each time step t :

$$\begin{aligned} h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h) \\ \hat{y}_t &= W_{hy}h_t + b_y \end{aligned}$$

where:

- $x_t \in \mathbb{R}^d$ is the input at time t
- $h_t \in \mathbb{R}^n$ is the hidden state at time t
- $W_{hh} \in \mathbb{R}^{n \times n}$ governs hidden-to-hidden transitions
- $W_{xh} \in \mathbb{R}^{n \times d}$ governs input-to-hidden connections
- $W_{hy} \in \mathbb{R}^{m \times n}$ governs hidden-to-output connections

Initialisation: $h_0 = \mathbf{0}$ (or learned initial state)

Unpacking the equations:

- The first equation updates the hidden state by combining the previous hidden state h_{t-1} with the current input x_t . The \tanh squashes the result to $(-1, 1)$.
- The second equation produces an output from the current hidden state.
- Both equations can be executed at each timestep, or outputs can be produced only at the end of the sequence (for classification).

Key insight: The **same weights** W_{hh}, W_{xh}, W_{hy} are used at every timestep. This is weight sharing across time, analogous to how CNNs share weights across space. The network learns a general transition function that works at any timestep.

156.2 Unrolled View and Backpropagation Through Time

To understand how RNNs are trained, we “unroll” the recurrent loop into a deep feedforward network.

Backpropagation Through Time (BPTT)

To train RNNs, “unroll” the network across timesteps, treating it as a deep feedforward network where each layer corresponds to a timestep:

$$\begin{array}{ccccccc} x_1 & \rightarrow & x_2 & \rightarrow & x_3 & \rightarrow & \cdots \\ \downarrow & & \downarrow & & \downarrow & & \\ h_1 & \rightarrow & h_2 & \rightarrow & h_3 & \rightarrow & \cdots \\ \downarrow & & \downarrow & & \downarrow & & \\ \hat{y}_1 & & \hat{y}_2 & & \hat{y}_3 & & \cdots \end{array}$$

Gradients are computed by backpropagating through this unrolled graph:

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_{t=1}^T \frac{\partial \mathcal{L}_t}{\partial W_{hh}} = \sum_{t=1}^T \sum_{k=1}^t \frac{\partial \mathcal{L}_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W_{hh}}$$

The term $\frac{\partial h_t}{\partial h_k}$ involves a product of $t - k$ Jacobians—this is where vanishing/exploding gradients arise.

Unpacking: Each loss \mathcal{L}_t depends on h_t , which depends on h_{t-1} , which depends on... all the way back to h_1 . To compute how \mathcal{L}_t depends on W_{hh} , we sum contributions through every path—every timestep where W_{hh} was used.

156.3 Vanishing and Exploding Gradients in RNNs

RNN Gradient Problem

The gradient through time involves:

$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^t \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=k+1}^t W_{hh}^\top \cdot \text{diag}(\sigma'(z_i))$$

where z_i is the pre-activation at step i .

If the spectral radius of W_{hh} is:

- < 1 : Gradients vanish exponentially with sequence length
- > 1 : Gradients explode exponentially with sequence length

Practical consequences:

- Vanilla RNNs struggle with sequences longer than $\sim 10\text{--}20$ steps
- Information from early timesteps cannot influence learning at later timesteps
- The network effectively has a “memory horizon” beyond which gradients are negligible

NB!

The vanishing gradient problem in RNNs is more severe than in feedforward networks because:

1. Sequences are often very long (hundreds or thousands of steps)
2. The *same* weight matrix W_{hh} is multiplied repeatedly, compounding the effect
3. Long-range dependencies are often semantically crucial (e.g., subject-verb agreement in text)

156.4 Long Short-Term Memory (LSTM)

LSTMs address the vanishing gradient problem through **gating mechanisms** and a separate **cell state**.

LSTM Architecture

An LSTM maintains two vectors: hidden state h_t and cell state c_t .

Gates (all in $[0, 1]$ via sigmoid activation):

$$\begin{aligned} f_t &= \sigma(W_f[h_{t-1}, x_t] + b_f) && \text{(forget gate)} \\ i_t &= \sigma(W_i[h_{t-1}, x_t] + b_i) && \text{(input gate)} \\ o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o) && \text{(output gate)} \end{aligned}$$

Candidate cell state:

$$\tilde{c}_t = \tanh(W_c[h_{t-1}, x_t] + b_c)$$

Cell state update:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

Hidden state output:

$$h_t = o_t \odot \tanh(c_t)$$

where \odot denotes element-wise multiplication and $[h_{t-1}, x_t]$ is concatenation.

Unpacking the gates:

- **Forget gate f_t :** Decides what proportion of the previous cell state to retain. Values near 1 preserve information; near 0 “forget” (reset) the cell state. Each element of c_{t-1} can be independently forgotten.
- **Input gate i_t :** Decides how much of the new candidate information to write to the cell state. Controls which new information is important.
- **Output gate o_t :** Decides how much of the cell state to expose as the hidden state. The cell state stores long-term memory; the hidden state is the “working memory” exposed to other layers.

Why LSTMs solve vanishing gradients: The cell state c_t can flow through time with only element-wise operations. When $f_t \approx 1$ and $i_t \approx 0$, the gradient through c_t is approximately 1:

$$\frac{\partial c_t}{\partial c_{t-1}} = f_t$$

If the forget gate stays open ($f_t \approx 1$), gradients flow unimpeded across arbitrarily long sequences. The network can learn to keep the forget gate open when long-term memory is needed.

156.5 Gated Recurrent Unit (GRU)

GRU Architecture

GRUs simplify LSTMs by merging cell and hidden states, using only two gates:

Gates:

$$\begin{aligned} z_t &= \sigma(W_z[h_{t-1}, x_t] + b_z) && \text{(update gate)} \\ r_t &= \sigma(W_r[h_{t-1}, x_t] + b_r) && \text{(reset gate)} \end{aligned}$$

Candidate hidden state:

$$\tilde{h}_t = \tanh(W_h[r_t \odot h_{t-1}, x_t] + b_h)$$

Hidden state update:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

Unpacking:

- **Update gate z_t :** Interpolates between old state and new candidate. When $z_t \approx 0$, the state is preserved (like LSTM forget gate ≈ 1). When $z_t \approx 1$, the state is replaced with the candidate.
- **Reset gate r_t :** Controls how much of the previous state to consider when computing the candidate. When $r_t \approx 0$, ignore history and behave like a feedforward network.

GRU vs LSTM:

- GRUs have fewer parameters (2 gates vs 3)
- Performance is similar on most tasks
- GRUs are faster to train
- LSTMs may have slight edge on tasks requiring very fine-grained memory control

156.6 Bidirectional RNNs

Bidirectional RNN

For tasks where future context is also available (e.g., tagging a complete sentence), process the sequence in both directions:

Forward pass: $\vec{h}_t = f(x_t, \vec{h}_{t-1})$

Backward pass: $\overleftarrow{h}_t = f(x_t, \overleftarrow{h}_{t+1})$

Combined representation: $h_t = [\vec{h}_t; \overleftarrow{h}_t]$ (concatenation)

This gives each position access to both past and future context.

NB!

Bidirectional RNNs cannot be used for autoregressive tasks (e.g., language generation) where future tokens are not available at inference time. They are appropriate for tasks like:

- Sequence labelling (POS tagging, NER)
- Sentence classification
- Machine translation encoding (but not decoding)

RNN Architecture Summary

- **Vanilla RNN:** Simple but suffers from vanishing gradients; limited to short sequences
- **LSTM:** Gated architecture with separate cell state; handles long-range dependencies
- **GRU:** Simplified LSTM; similar performance with fewer parameters
- **Bidirectional:** Access to past and future context; cannot be used for generation

157 Attention Mechanisms

Attention: Key Idea

Attention computes a **weighted average** of values, where weights are determined by the relevance of each value to a query. It enables:

- Flexible, learned representations
- Direct access to all positions (no sequential bottleneck)
- Interpretable importance weights

Attention is perhaps the most important architectural innovation in modern deep learning. It allows networks to dynamically focus on relevant parts of the input, rather than relying on fixed patterns (CNNs) or sequential accumulation (RNNs).

157.1 Motivation: The Bottleneck Problem

In sequence-to-sequence models (e.g., machine translation), an encoder RNN compresses the entire input into a single fixed-length vector, which the decoder must use to generate the output.

Problem: For long sequences, this bottleneck loses information. The final hidden state cannot capture all nuances of a 100-word sentence. Important details from early in the sentence may be “washed out” by later processing.

Solution: Instead of using only the final state, let the decoder *attend* to all encoder states, focusing on relevant parts for each output token. When generating the French word for “cat”, look at the English representation of “cat”, not just the final state that summarises the whole sentence.

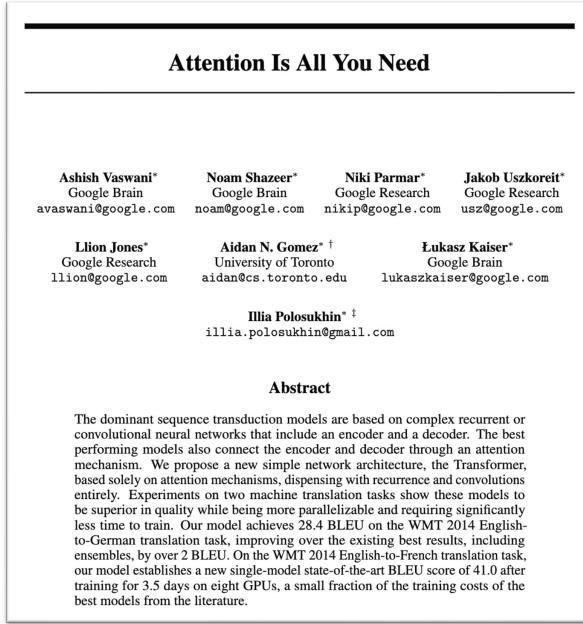


Figure 104: Attention mechanism: queries attend to keys, retrieving weighted combinations of values.

157.2 General Attention

Attention Definition

Given:

- Query q (what we're looking for)
- Keys k_1, \dots, k_n (features describing each value)
- Values v_1, \dots, v_n (content to retrieve)
- Similarity function $\phi(\cdot, \cdot)$

Attention output:

$$\text{Attn}(q, \{k_i, v_i\}) = \sum_{i=1}^n \alpha_i \cdot v_i$$

where the attention weights are:

$$\alpha_i = \frac{\phi(q, k_i)}{\sum_{j=1}^n \phi(q, k_j)}$$

The weights α_i sum to 1, forming a probability distribution over values.

Unpacking the components:

- **Query:** Represents “what am I looking for?”—the question being asked
- **Keys:** Represent “what does each position contain?”—descriptors for matching
- **Values:** Represent “what should I retrieve?”—the actual content

- **Similarity ϕ :** Measures how well the query matches each key
- **Output:** Weighted average of values, where weights come from query-key similarities

Interpretation as soft dictionary lookup: A standard dictionary returns exactly one value for an exact key match. Attention is a “soft” lookup—it returns a weighted combination based on similarity. If multiple keys partially match, their values all contribute proportionally.

Common similarity functions:

- **Dot product:** $\phi(q, k) = q^\top k$ —simple, fast, but magnitude-sensitive
- **Scaled dot product:** $\phi(q, k) = \frac{q^\top k}{\sqrt{d}}$ —normalises for dimension
- **Additive (Bahdanau):** $\phi(q, k) = v^\top \tanh(W_q q + W_k k)$ —more flexible, separate projections

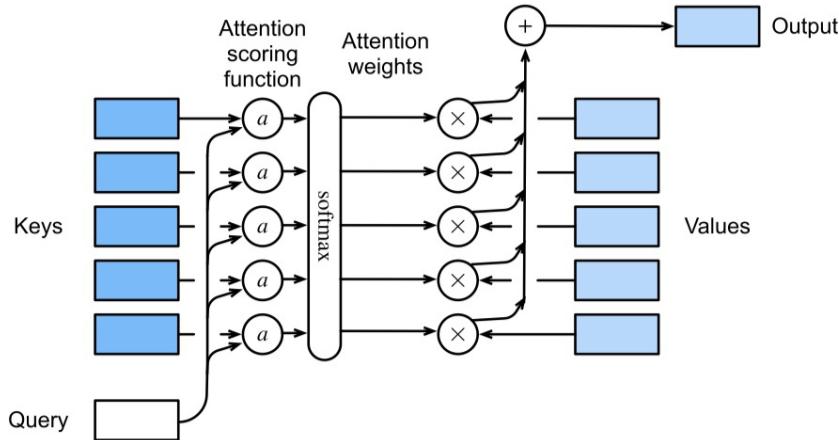


Figure 105: Attention weights visualised: each query attends differently to the available keys.

157.3 Scaled Dot-Product Attention

The most common attention implementation, used in Transformers.

Scaled Dot-Product Attention

For query matrix $Q \in \mathbb{R}^{m \times d_k}$, key matrix $K \in \mathbb{R}^{n \times d_k}$, value matrix $V \in \mathbb{R}^{n \times d_v}$:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

Steps:

1. Compute similarity: $QK^\top \in \mathbb{R}^{m \times n}$ (dot products between all query-key pairs)
2. Scale by $\sqrt{d_k}$ to prevent large values
3. Apply softmax row-wise to get attention weights $\in \mathbb{R}^{m \times n}$
4. Multiply by V to get weighted values $\in \mathbb{R}^{m \times d_v}$

Unpacking the matrix dimensions:

- Q has m queries, each d_k -dimensional
- K has n keys, each d_k -dimensional
- V has n values, each d_v -dimensional
- QK^\top gives an $m \times n$ matrix of all query-key similarities
- After softmax, each row sums to 1 (attention distribution for each query)
- Multiplying by V gives m outputs, each a weighted average of values

157.3.1 Why Scale by $\sqrt{d_k}$?

This scaling is crucial for stable training.

If query and key elements are approximately standard normal, their dot product has variance proportional to d_k :

$$\text{Var}(q^\top k) = \sum_{i=1}^{d_k} \text{Var}(q_i k_i) = d_k \cdot 1 = d_k$$

Problem: For large d_k (e.g., 512), dot products can be very large in magnitude. Softmax of large values saturates to near-one-hot distributions, where gradients are extremely small.

Solution: Dividing by $\sqrt{d_k}$ normalises variance to 1, keeping dot products in a reasonable range where softmax gradients are healthy.

157.3.2 Softmax

Row-wise Softmax

For matrix $X \in \mathbb{R}^{m \times n}$:

$$\text{softmax}(X)_{ij} = \frac{\exp(x_{ij})}{\sum_{k=1}^n \exp(x_{ik})}$$

Each row sums to 1, converting raw scores to a probability distribution over columns (keys).

157.4 Multi-Head Attention

Multi-Head Attention

Instead of a single attention function, apply attention multiple times in parallel with different learned projections:

For each head $i \in \{1, \dots, h\}$:

$$Q_i = QW_i^Q, \quad K_i = KW_i^K, \quad V_i = VW_i^V \\ \text{head}_i = \text{Attention}(Q_i, K_i, V_i)$$

Concatenate and project:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

where $W_i^Q, W_i^K \in \mathbb{R}^{d \times d_k}$, $W_i^V \in \mathbb{R}^{d \times d_v}$, and $W^O \in \mathbb{R}^{hd_v \times d}$.

Why multiple heads?

- Different heads can attend to different types of relationships
- One head might capture syntactic dependencies (subject-verb), another semantic similarity (synonyms)
- Analogous to using multiple filters in a CNN layer—each head learns a different “view”

Typically $d_k = d_v = d/h$ so that the total computation is similar to single-head attention with full dimensionality.

157.5 Self-Attention

Self-attention allows each position in a sequence to attend to all other positions, capturing dependencies regardless of distance.

Self-Attention

Given input $X \in \mathbb{R}^{n \times d}$ (sequence of n vectors of dimension d), compute queries, keys, and values via learned projections:

$$\begin{aligned} Q &= XW_Q \\ K &= XW_K \\ V &= XW_V \end{aligned}$$

Then apply scaled dot-product attention:

$$\text{SelfAttn}(X) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

The output has the same shape as the input: n vectors of dimension d_v .

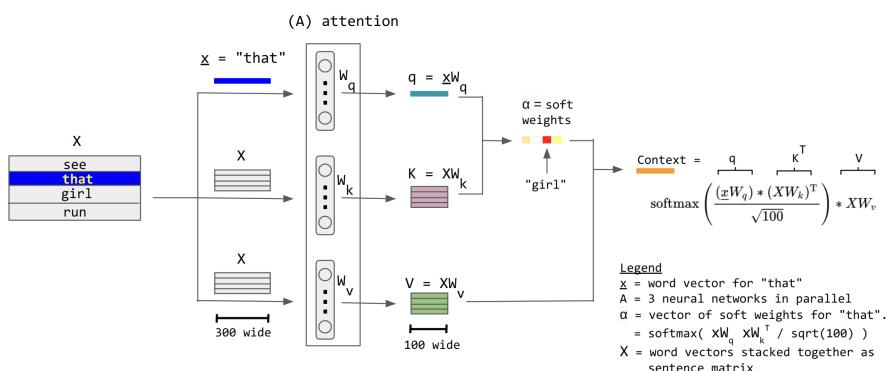


Figure 106: Self-attention: each position computes attention over all positions, enabling global context.

What self-attention learns:

1. W_Q : How to represent “what I’m looking for”
2. W_K : How to represent “what I contain”
3. W_V : How to represent “what I provide when attended to”

Connection to kernel regression: Self-attention is essentially kernel regression where:

- The kernel (similarity function) is learned
- The input representation is learned
- The output representation is learned

This is extremely flexible (and over-parameterised)—we're simultaneously learning what to compare, how to compare, and what to output.

Why Self-Attention?

- **Global context:** Every position can directly attend to every other position (path length 1)
- **Parallelisable:** No sequential dependencies (unlike RNNs)
- **Flexible:** Learns which relationships matter, rather than assuming fixed patterns
- **Interpretable:** Attention weights show which positions influence each output

158 Transformers

Transformers: Key Idea

Transformers replace recurrence entirely with self-attention:

- Process all positions in parallel
- Capture long-range dependencies in a single layer
- Scale efficiently to very long sequences and large models

Result: State-of-the-art performance across NLP, vision, and beyond.

The Transformer architecture (Vaswani et al., 2017, “Attention Is All You Need”) revolutionised deep learning by showing that attention alone, without recurrence or convolution, achieves superior performance on sequence tasks.

158.1 Architecture Overview

The original Transformer follows an encoder-decoder structure for sequence-to-sequence tasks.

Transformer Architecture

Encoder (processes input sequence):

1. Input embedding + positional encoding
2. $N \times$ encoder layers, each containing:
 - Multi-head self-attention
 - Add & layer normalisation (residual connection)
 - Position-wise feed-forward network
 - Add & layer normalisation

Decoder (generates output sequence):

1. Output embedding + positional encoding
2. $N \times$ decoder layers, each containing:
 - Masked multi-head self-attention (prevents attending to future positions)
 - Add & layer normalisation
 - Multi-head cross-attention (attends to encoder outputs)
 - Add & layer normalisation
 - Position-wise feed-forward network
 - Add & layer normalisation

Output: Linear projection + softmax over vocabulary

Unpacking the components:

- **Residual connections:** Every sub-layer has a skip connection, as in ResNets
- **Layer normalisation:** Stabilises training (discussed below)
- **Encoder self-attention:** Each position attends to all positions in the input
- **Decoder self-attention:** Each position attends only to earlier positions (causal)
- **Cross-attention:** Decoder positions attend to encoder outputs

158.2 Positional Encoding

Why Positional Encoding?

Self-attention is **permutation equivariant**: if we shuffle the input sequence, the output shuffles identically. This means the model has no inherent notion of position—“cat sat on mat” and “mat on sat cat” produce the same attention pattern.

Solution: Add positional information to the input embeddings.

Sinusoidal Positional Encoding

For position pos and dimension i :

$$\begin{aligned} PE_{(\text{pos},2i)} &= \sin\left(\frac{\text{pos}}{10000^{2i/d}}\right) \\ PE_{(\text{pos},2i+1)} &= \cos\left(\frac{\text{pos}}{10000^{2i/d}}\right) \end{aligned}$$

Properties:

- Each position has a unique encoding
- Relative positions can be represented as linear functions of the encodings
- Generalises to sequence lengths not seen during training

Intuition: The positional encoding acts like a binary counter where different dimensions oscillate at different frequencies. Low-frequency dimensions (large i) distinguish distant positions; high-frequency dimensions (small i) distinguish nearby positions. The model can learn to use these patterns to understand position.

Alternative: Modern models often use **learned positional embeddings** instead—a separate embedding vector for each position, learned during training. This works well when maximum sequence length is known and fixed.

158.3 Layer Normalisation

Layer Normalisation

Unlike batch normalisation (which normalises across the batch), layer normalisation normalises across features for each example:

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

where μ and σ^2 are the mean and variance computed across the feature dimension (not the batch dimension).

Advantages over batch normalisation:

- Works with batch size 1
- No running statistics needed at test time
- Better suited for variable-length sequences

Key difference: Batch norm computes statistics over examples (same feature across batch). Layer norm computes statistics over features (same example across dimensions). For sequences with varying lengths and small batch sizes, layer norm is more stable and consistent.

158.4 Position-wise Feed-Forward Networks

Feed-Forward Block

Each position is processed independently by the same two-layer network:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Typically the inner dimension is $4 \times$ the model dimension (e.g., $d = 512$, inner = 2048).

This can be seen as two 1×1 convolutions—the same transformation applied at every position.

Purpose: The attention layer mixes information between positions but applies only linear transformations. The FFN provides position-wise nonlinearity, allowing the model to transform each position's representation in complex ways.

158.5 Masked Self-Attention

Causal Masking

In the decoder, each position should only attend to earlier positions (not “see the future”). This is enforced by masking:

$$\text{MaskedAttn}(Q, K, V) = \text{softmax} \left(\frac{QK^\top}{\sqrt{d_k}} + M \right) V$$

where M is a mask matrix with $M_{ij} = 0$ if $i \geq j$ (allowed) and $M_{ij} = -\infty$ if $i < j$ (blocked).

The $-\infty$ values become 0 after softmax, preventing information flow from future positions.

Why masking? During training, we process all positions in parallel. But during generation, position t should not know what comes at positions $t+1, t+2, \dots$. The mask ensures the training-time behaviour matches the inference-time constraint.

158.6 Why Transformers Replaced RNNs

Computational Comparison

Model	Sequential Ops	Path Length	Complexity/Layer
RNN	$O(n)$	$O(n)$	$O(n \cdot d^2)$
Transformer	$O(1)$	$O(1)$	$O(n^2 \cdot d)$

- **Sequential operations:** RNNs must process tokens sequentially; Transformers process all in parallel
- **Maximum path length:** Information travels $O(n)$ steps in RNNs but only $O(1)$ in Transformers
- **Per-layer complexity:** Transformers have $O(n^2)$ attention cost, but this is highly parallelisable

Key advantages of Transformers:

1. **Parallelisation:** Training is much faster on modern hardware (GPUs/TPUs) because all positions can be computed simultaneously
2. **Long-range dependencies:** Direct connections between any two positions in a single layer
3. **Scalability:** Performance improves with more data, compute, and parameters (“scaling laws”)
4. **Transfer learning:** Pre-trained models transfer well to downstream tasks

NB!

The $O(n^2)$ attention complexity is a limitation for very long sequences. For a 10,000-token document, the attention matrix has 100 million entries. Various “efficient Transformer” variants address this (Longformer, BigBird, etc.), but the quadratic scaling remains a fundamental constraint.

158.7 Pre-training and Transfer Learning

The Pre-training Paradigm

Pre-training: Train a large model on a massive unlabelled corpus with a self-supervised objective:

- **Masked language modelling (BERT):** Predict masked tokens from context
- **Causal language modelling (GPT):** Predict the next token
- **Denoising (T5):** Reconstruct corrupted text

Fine-tuning: Adapt the pre-trained model to a specific task with a small labelled dataset.

Key insight: Language understanding emerges from predicting text at scale. The resulting representations transfer to almost any NLP task.

Why this works:

- Predicting language requires understanding syntax, semantics, and world knowledge
- Billions of training examples enable learning rich representations
- Fine-tuning requires only thousands of examples for many tasks
- The same architecture scales from small to very large models

Transformer Impact

Transformers have become the dominant architecture for:

- **NLP:** BERT, GPT, T5, and their successors
- **Vision:** Vision Transformers (ViT), treating image patches as tokens
- **Multimodal:** CLIP, Flamingo, combining vision and language
- **Generative AI:** Large language models, image generation (diffusion with Transformers)

159 Practical Considerations

Choosing an Architecture

Use CNNs when:

- Data has spatial structure (images, spectrograms)
- Translation invariance is desired
- Computational efficiency is critical

Use RNNs/LSTMs when:

- Sequences are short to moderate length
- Streaming/online processing is required
- Memory constraints prevent attention over full sequence

Use Transformers when:

- Long-range dependencies are important
- Pre-trained models are available for your domain
- Compute resources allow attention over the sequence
- State-of-the-art performance is the priority

159.1 Transfer Learning in Practice

Transfer Learning Strategies

Feature extraction:

- Freeze pre-trained weights
- Train only new task-specific layers
- Fast, requires minimal data

Fine-tuning:

- Initialise with pre-trained weights
- Train all parameters (often with lower learning rate for pre-trained layers)
- Better performance, requires more data and compute

Layer-wise learning rates:

- Lower learning rates for earlier layers (more general features)
- Higher learning rates for later layers (more task-specific)
- Prevents catastrophic forgetting of pre-trained knowledge

When transfer learning helps most:

- Target task has limited labelled data
- Source and target domains are related
- Pre-trained model is large and well-trained

159.2 Modern Best Practices

Training Deep Networks

Architecture:

- Use residual connections for networks deeper than ~ 10 layers
- Prefer layer normalisation for Transformers, batch normalisation for CNNs
- Use dropout for regularisation (typically 0.1–0.3)

Optimisation:

- Adam or AdamW (weight decay variant) as default optimiser
- Learning rate warmup followed by decay (linear, cosine, or step)
- Gradient clipping for RNNs and Transformers

Regularisation:

- Data augmentation is often the most effective regulariser
- Weight decay (L2) is standard
- Dropout in fully connected layers, sometimes in attention

Training stability:

- Monitor loss curves and gradient norms
- Use mixed-precision training (FP16) for efficiency
- Start with a working configuration, then tune

160 Summary

Key Concepts from Week 11

1. **Gradient problems:** Vanishing (use ReLU, residual connections, LSTM/GRU); Exploding (use gradient clipping)
2. **Batch normalisation:** Stabilises activations, enables higher learning rates
3. **Regularisation:** Weight decay constrains magnitudes; dropout encourages redundancy
4. **CNNs:** Exploit local structure via convolutions; weight sharing provides translation equivariance; pooling provides translation invariance
5. **Residual connections:** Enable training of very deep networks by providing gradient shortcuts
6. **RNNs:** Maintain hidden state for sequential data; vanilla RNNs suffer from vanishing gradients
7. **LSTM/GRU:** Gated architectures that preserve gradients over long sequences
8. **Attention:** Learned weighted averaging; enables flexible, global context
9. **Self-attention:** Each position attends to all positions; foundation of Transformers
10. **Transformers:** Replace recurrence with attention; parallelisable and scalable
11. **Transfer learning:** Pre-train on large data, fine-tune on specific tasks

Architecture Decision Guide

Data Type	Recommended Architecture
Images	CNN (or ViT if pre-trained model available)
Short sequences (< 100)	LSTM/GRU or Transformer
Long sequences (> 100)	Transformer
Text with pre-training	Transformer (BERT, GPT, etc.)
Streaming/online data	RNN/LSTM (process one token at a time)
Tabular data	MLP (potentially with attention)

161 Overview

Chapter Summary: Unsupervised Learning

This chapter develops the theory and practice of **unsupervised learning**—finding structure in data without labels:

- **Dimensionality reduction:** PCA, factor analysis, ICA
- **Manifold learning:** t-SNE, UMAP, and the manifold hypothesis
- **Clustering:** K-means, hierarchical methods, DBSCAN, Gaussian mixture models
- **Representation learning:** Autoencoders, variational autoencoders
- **Self-supervised learning:** Contrastive methods and pretext tasks

Unifying theme: Compression and structure discovery—finding low-dimensional representations that preserve meaningful information.

161.1 What is Unsupervised Learning?

In supervised learning, we have input-output pairs (x_i, y_i) and seek to learn the mapping $x \mapsto y$. In unsupervised learning, we have only inputs $\{x_1, \dots, x_n\}$ and seek to discover **structure** in the data itself. There is no “correct answer” to check against—the data must speak for itself. A useful mental model: unsupervised learning is sometimes called “supervised learning in a trenchcoat.” Many unsupervised methods can be reframed as predicting something about the data from the data itself. PCA predicts the input from a compressed version of itself. Autoencoders do the same with neural networks. Clustering predicts which group a point belongs to. The key insight is that **structure enables prediction**, so finding structure is equivalent to finding predictable patterns.

Core Tasks in Unsupervised Learning

1. Dimensionality reduction: Find a low-dimensional representation $z_i \in \mathbb{R}^L$ for each $x_i \in \mathbb{R}^D$ where $L \ll D$, preserving relevant structure.

What this means: High-dimensional data (like images with thousands of pixels) often has an underlying structure that can be described with far fewer numbers. A photo of a face might have millions of pixels, but we can describe its essential content with a handful of parameters: age, gender, expression, lighting direction, pose angle. Dimensionality reduction finds these compact descriptions automatically.

2. Clustering: Partition data into groups such that points within a group are more similar to each other than to points in other groups.

What this means: The assumption is that data naturally forms discrete groups or categories. Customer segments, document topics, species of flowers—clustering discovers these groupings without being told what to look for.

3. Density estimation: Model the probability distribution $p(x)$ from which the data were drawn.

What this means: Instead of finding a single representation or partition, we model the entire distribution. This enables generation of new samples, anomaly detection (low probability regions), and understanding what “typical” data looks like.

4. Representation learning: Learn features that are useful for downstream tasks (transfer learning, visualisation, compression).

What this means: Often we don’t know what task we’ll eventually care about. Representation learning finds general-purpose features that capture meaningful structure, which can then be reused for many purposes.

These tasks are interrelated: clustering assumes a mixture model (density estimation), autoencoders perform dimensionality reduction while learning representations, and so forth.

161.2 Why is Unsupervised Learning Hard?

NB!

Unsupervised learning lacks the clear objective that supervised learning enjoys. Key challenges:

- **No ground truth:** Without labels, we cannot directly measure “correctness”
- **Ill-defined objectives:** What makes one clustering “better” than another? It depends on the application
- **Evaluation difficulty:** Metrics like reconstruction error or cluster compactness may not correlate with usefulness
- **The curse of dimensionality:** High-dimensional data is sparse; notions of distance and density become unreliable

Despite these challenges, unsupervised learning is essential—most real-world data is unlabelled, and discovering structure is often the first step in understanding a new domain.

The key insight is that unsupervised learning requires **inductive biases**—assumptions about what kinds of structure are meaningful. PCA assumes linear structure matters; clustering assumes discrete groups exist; autoencoders assume smooth, compressible representations exist. Different methods encode different assumptions.

Why does this matter? Consider clustering: there are infinitely many ways to partition a dataset. K-means assumes clusters are roughly spherical and equally sized. DBSCAN assumes clusters are dense regions separated by sparse regions. Neither is “correct”—they encode different assumptions that are appropriate for different problems. Choosing an algorithm is choosing an inductive bias.

162 Principal Component Analysis

Section Summary: PCA

- **Goal:** Find orthogonal directions of maximum variance
- **Two equivalent views:** Maximise variance \Leftrightarrow minimise reconstruction error
- **Solution:** Eigenvectors of the covariance matrix
- **Key result:** First L principal components give the best L -dimensional linear approximation
- **Practical use:** Preprocessing, visualisation, noise reduction, feature extraction

162.1 Motivation

High-dimensional data often lies near a lower-dimensional subspace. Consider D -dimensional data where most variation occurs along a few directions—the remaining dimensions contain mostly noise. PCA finds these directions of maximum variation.

Example: Handwritten digit images might have 784 pixels (28×28), but the “space of plausible digits” is much smaller. Strokes, curves, and angles can be described with far fewer parameters. Most combinations of 784 pixel values would look like random static, not recognisable digits. The actual digits occupy a tiny, structured subset of the full 784-dimensional space.

Geometric intuition: Imagine your data points form an elongated cloud in 3D space—like a cigar shape. Most of the variation is along the cigar’s length, with much less variation in the two perpendicular directions. PCA finds that the cigar’s long axis is the first principal component (most variance), and the two short axes are the second and third components (less variance). If we project onto just the first component, we capture most of the structure while reducing from 3D to 1D.

162.2 Two Equivalent Formulations

PCA can be derived from two perspectives that yield identical solutions. Understanding both provides complementary intuition.

Formulation 1: Maximum Variance

Objective: Find the direction $w_1 \in \mathbb{R}^D$ along which projections have maximum variance.

Given centred data $\{x_1, \dots, x_n\}$ (i.e., $\sum_i x_i = 0$), the projection of x_i onto unit vector w is $z_i = w^\top x_i$. The variance of projections is:

$$\text{Var}(z) = \frac{1}{n} \sum_{i=1}^n (w^\top x_i)^2 = \frac{1}{n} \sum_{i=1}^n w^\top x_i x_i^\top w = w^\top \left(\frac{1}{n} X^\top X \right) w = w^\top \Sigma w$$

where $\Sigma = \frac{1}{n} X^\top X$ is the sample covariance matrix (for centred data).

Breaking down the algebra:

- $w^\top x_i$ is a scalar: the projection of point x_i onto direction w
- $(w^\top x_i)^2 = (w^\top x_i)(x_i^\top w) = w^\top (x_i x_i^\top) w$ using the fact that $w^\top x_i$ is a scalar
- Summing over all points and pulling w outside: $w^\top (\frac{1}{n} \sum_i x_i x_i^\top) w = w^\top \Sigma w$
- The matrix $\Sigma = \frac{1}{n} X^\top X$ is the covariance matrix of centred data

Optimisation problem:

$$w_1 = \arg \max_{\|w\|=1} w^\top \Sigma w$$

The constraint $\|w\| = 1$ is essential—otherwise we could make the variance arbitrarily large by scaling w .

Subsequent components w_2, w_3, \dots maximise variance subject to orthogonality: $w_k \perp w_1, \dots, w_{k-1}$.

The maximum variance view asks: “If I can only look at one direction, which direction shows me the most variation in my data?” Then: “Given I’ve already captured that, which orthogonal direction shows the next most variation?”

Formulation 2: Minimum Reconstruction Error

Objective: Find an L -dimensional subspace such that projecting onto it and back minimises squared reconstruction error.

Let $W \in \mathbb{R}^{D \times L}$ be a matrix of orthonormal columns spanning the subspace. The projection of x is $\hat{x} = WW^\top x$. The reconstruction error is:

$$\mathcal{L}(W) = \frac{1}{n} \sum_{i=1}^n \|x_i - WW^\top x_i\|^2$$

Understanding the reconstruction:

- $W^\top x_i \in \mathbb{R}^L$ gives the coordinates of x_i in the L -dimensional subspace
- $W(W^\top x_i) \in \mathbb{R}^D$ reconstructs from those coordinates back to the original space
- WW^\top is a projection matrix: it projects any vector onto the subspace spanned by W 's columns
- $\|x_i - WW^\top x_i\|^2$ measures how much information is lost in this projection

Optimisation problem:

$$W^* = \arg \min_{W^\top W = I_L} \frac{1}{n} \sum_{i=1}^n \|x_i - WW^\top x_i\|^2$$

Equivalence: These formulations yield the same solution because

$$\|x_i - WW^\top x_i\|^2 = \|x_i\|^2 - \|W^\top x_i\|^2$$

Derivation of equivalence:

$$\begin{aligned} \|x_i - WW^\top x_i\|^2 &= (x_i - WW^\top x_i)^\top (x_i - WW^\top x_i) \\ &= x_i^\top x_i - 2x_i^\top WW^\top x_i + x_i^\top WW^\top WW^\top x_i \\ &= \|x_i\|^2 - 2\|W^\top x_i\|^2 + \|W^\top x_i\|^2 \quad (\text{since } W^\top W = I) \\ &= \|x_i\|^2 - \|W^\top x_i\|^2 \end{aligned}$$

Since $\|x_i\|^2$ is fixed, minimising reconstruction error is equivalent to maximising $\sum_i \|W^\top x_i\|^2$, the variance of projections.

The minimum reconstruction error view asks: “If I compress my data to L dimensions and then try to recover the original, how can I minimise the information loss?” The answer is the same: project onto the directions of maximum variance.

162.3 Solution via Eigendecomposition

Derivation of PCA Solution

To solve $\max_{\|w\|=1} w^\top \Sigma w$, form the Lagrangian:

$$\mathcal{L}(w, \lambda) = w^\top \Sigma w - \lambda(w^\top w - 1)$$

What this means: The Lagrangian incorporates the constraint $\|w\|^2 = w^\top w = 1$ into the objective. The multiplier λ will turn out to have a meaningful interpretation.

Taking the gradient and setting to zero:

$$\nabla_w \mathcal{L} = 2\Sigma w - 2\lambda w = 0 \quad \Rightarrow \quad \Sigma w = \lambda w$$

Key recognition: This is the eigenvalue equation! The solution w must be an eigenvector of Σ , and λ is the corresponding eigenvalue.

Which eigenvector? Substituting back:

$$w^\top \Sigma w = w^\top (\lambda w) = \lambda \|w\|^2 = \lambda$$

The variance equals the eigenvalue. To maximise variance, choose the eigenvector with the **largest eigenvalue**.

General solution: Order eigenvalues $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_D \geq 0$ with corresponding eigenvectors v_1, v_2, \dots, v_D . The first L principal components are v_1, \dots, v_L .

Why orthogonality comes for free: The covariance matrix Σ is symmetric (since $\Sigma = \frac{1}{n}X^\top X$). Symmetric matrices have orthogonal eigenvectors—this is a fundamental result from linear algebra. So the principal components are automatically orthogonal without us needing to enforce it explicitly.

Key Properties of PCA Solution

1. The covariance matrix Σ is symmetric positive semi-definite, so eigenvalues are real and non-negative
2. Eigenvectors are orthogonal: $v_i^\top v_j = 0$ for $i \neq j$
3. Total variance is preserved: $\sum_{l=1}^D \lambda_l = \text{tr}(\Sigma) = \sum_{d=1}^D \text{Var}(X_d)$
4. Variance explained by first L components: $\frac{\sum_{l=1}^L \lambda_l}{\sum_{l=1}^D \lambda_l}$

Intuition for property 3: The total variance in your data is fixed—it's just redistributed among the principal components. The trace of the covariance matrix equals the sum of variances along each original axis, which equals the sum of eigenvalues. PCA doesn't create or destroy variance; it reorganises it so that maximum variance is concentrated in the first few components.

162.4 The PCA Algorithm

PCA Algorithm

Input: Data matrix $X \in \mathbb{R}^{n \times D}$, number of components L

Output: Principal components $W \in \mathbb{R}^{D \times L}$, projected data $Z \in \mathbb{R}^{n \times L}$

1. **Centre the data:** $\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$, then $\tilde{X} = X - \mathbf{1}\bar{x}^\top$

What this does: Subtracts the mean from each feature so that the centred data has zero mean. The matrix $\mathbf{1}\bar{x}^\top$ has identical rows, each equal to the mean vector.

2. **Compute covariance matrix:** $\Sigma = \frac{1}{n} \tilde{X}^\top \tilde{X}$

What this does: Creates a $D \times D$ matrix where entry (i, j) is the covariance between features i and j .

3. **Eigendecomposition:** Compute eigenpairs (λ_l, v_l) of Σ

What this does: Finds the directions (eigenvectors) and variances (eigenvalues) of the principal components.

4. **Select top L eigenvectors:** $W = [v_1 | v_2 | \dots | v_L]$

What this does: Keeps only the L directions with highest variance.

5. **Project data:** $Z = \tilde{X}W$

What this does: Computes the L -dimensional representation of each data point.

Reconstruction: $\hat{X} = ZW^\top + \mathbf{1}\bar{x}^\top$

What this does: Recovers an approximation to the original data by mapping back from the low-dimensional representation and adding back the mean.

NB!

Common mistakes with PCA:

- **Forgetting to centre:** PCA assumes zero-mean data; failing to centre gives incorrect results. The first “principal component” would point toward the mean rather than the direction of maximum variance.
- **Not scaling features:** If features have different units/scales, PCA will be dominated by high-variance features. Consider standardising (mean 0, variance 1) when features are not comparable. Example: if one feature is height in metres (range 1.5–2.0) and another is income in pounds (range 20000–100000), the income feature will dominate all principal components.
- **Interpreting components as features:** Principal components are linear combinations—they may not have intuitive meaning. PC1 might be “0.3 × height + 0.7 × income + ...”, which is hard to interpret.
- **Assuming linearity suffices:** PCA only captures linear structure; nonlinear manifolds require different methods (kernel PCA, autoencoders, t-SNE).

162.5 Choosing the Number of Components

A critical practical question: how many principal components should we keep?

Methods for Selecting L

- Cumulative explained variance:** Choose L such that $\frac{\sum_{l=1}^L \lambda_l}{\sum_{l=1}^D \lambda_l} \geq \tau$ for some threshold τ (e.g., 0.95).

Intuition: Keep enough components to explain 95% (or 90%, or 99%) of the total variance. The remaining components explain mostly noise.

- Scree plot:** Plot eigenvalues λ_l vs l . Look for an “elbow” where eigenvalues drop sharply—components beyond the elbow contribute little.

Intuition: The eigenvalues often decay smoothly, but sometimes there’s a clear break between “signal” components (large eigenvalues) and “noise” components (small eigenvalues). The elbow marks this transition.

- Kaiser criterion:** Keep components with $\lambda_l > \bar{\lambda}$ (eigenvalue above average). For standardised data, this means $\lambda_l > 1$.

Intuition: If a component explains less variance than the average original feature, it’s not capturing meaningful structure.

- Cross-validation:** If using PCA for a downstream task, choose L that optimises performance on held-out data.

Intuition: Let the task tell you how much compression is acceptable. If 10 components give the same classification accuracy as 50, prefer 10.

- Parallel analysis:** Compare eigenvalues to those from random data with the same dimensions. Keep components with eigenvalues significantly larger than random.

Intuition: Random noise also produces eigenvalues, but they follow a predictable distribution. Only keep components whose eigenvalues exceed what chance would produce.

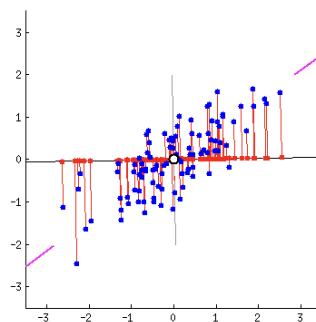


Figure 107: PCA projection: data points (blue) projected onto the first principal component (red line). The principal component direction captures maximum variance; projections minimise perpendicular distance to this line. The residuals (perpendicular distances from points to the line) represent the reconstruction error.

162.6 PCA and Singular Value Decomposition

In practice, PCA is usually computed via SVD rather than eigendecomposition, for both numerical and computational reasons.

SVD Formulation of PCA

The singular value decomposition of the centred data matrix $\tilde{X} \in \mathbb{R}^{n \times D}$ is:

$$\tilde{X} = USV^\top$$

where:

- $U \in \mathbb{R}^{n \times r}$: left singular vectors (orthonormal columns)
- $S \in \mathbb{R}^{r \times r}$: diagonal matrix of singular values $s_1 \geq s_2 \geq \dots \geq s_r > 0$
- $V \in \mathbb{R}^{D \times r}$: right singular vectors (orthonormal columns)
- $r = \text{rank}(\tilde{X}) \leq \min(n, D)$

Connection to PCA:

$$\Sigma = \frac{1}{n} \tilde{X}^\top \tilde{X} = \frac{1}{n} VSU^\top USV^\top = V \left(\frac{S^2}{n} \right) V^\top$$

Breaking this down:

- $U^\top U = I$ because U has orthonormal columns
- So $\tilde{X}^\top \tilde{X} = VS^2V^\top$
- This is the eigendecomposition of $\tilde{X}^\top \tilde{X}$: V are eigenvectors, S^2 are eigenvalues

Thus:

- The right singular vectors V are the principal components
- The eigenvalues of Σ are $\lambda_l = s_l^2/n$
- The projected data is $Z = \tilde{X}V = US$

Why Use SVD for PCA?

- **Numerical stability:** SVD is more stable than forming $\tilde{X}^\top \tilde{X}$ (which squares condition number). If your data has features on very different scales, forming the covariance matrix can lose precision.
- **Efficiency for tall matrices:** When $n \gg D$, compute $\tilde{X}^\top \tilde{X}$ ($D \times D$). When $n \ll D$ (more features than samples, common in genomics), compute $\tilde{X} \tilde{X}^\top$ ($n \times n$) instead.
- **Truncated SVD:** Can compute only the top L singular vectors without full decomposition (important for large D). Libraries like `sklearn.decomposition.TruncatedSVD` use randomised algorithms that are much faster.
- **Direct computation:** Avoid explicitly forming covariance matrix, which can be memory-intensive for large D .

162.7 Kernel PCA

PCA finds linear projections, but data may have nonlinear structure. **Kernel PCA** applies PCA in a high-dimensional feature space implicitly defined by a kernel function.

Kernel PCA

Idea: Map data via $\phi : \mathbb{R}^D \rightarrow \mathcal{H}$ to a (possibly infinite-dimensional) feature space and perform PCA there.

Problem: We cannot explicitly compute $\phi(x)$ for kernels like the RBF. The feature space might be infinite-dimensional!

Solution: Express PCA in terms of dot products only, then replace with kernel evaluations. This is the kernel trick (see Week 6: Kernel Methods).

In feature space, the covariance matrix is $C = \frac{1}{n} \sum_{i=1}^n \phi(x_i)\phi(x_i)^\top$. Principal components v satisfy $Cv = \lambda v$.

Key insight: v lies in the span of the data:

$$v = \sum_{i=1}^n \alpha_i \phi(x_i)$$

Why? Any component orthogonal to the data has zero variance, so it cannot be a principal component. This is the representer theorem in action.

Substituting and taking inner products with $\phi(x_j)$:

$$K\alpha = n\lambda\alpha$$

where $K_{ij} = k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$ is the kernel matrix.

Algorithm:

1. Centre the kernel matrix: $\tilde{K} = K - \frac{1}{n}\mathbf{1}K - \frac{1}{n}K\mathbf{1} + \frac{1}{n^2}\mathbf{1}\mathbf{1}^\top$
2. Compute eigenvectors $\alpha^{(l)}$ of \tilde{K}
3. Project new point x as: $z_l = \sum_{i=1}^n \alpha_i^{(l)} k(x_i, x)$

Common kernels:

- RBF/Gaussian: $k(x, y) = \exp(-\gamma\|x - y\|^2)$ — infinite-dimensional feature space
- Polynomial: $k(x, y) = (x^\top y + c)^d$ — captures polynomial interactions

Kernel PCA can capture nonlinear structure (e.g., unrolling a Swiss roll), but choosing the kernel and its parameters requires care. See Week 6 (Kernel Methods) for kernel theory and the kernel trick.

163 Other Linear Dimensionality Reduction Methods

Section Summary: Beyond PCA

- **Factor Analysis:** Probabilistic model assuming latent factors plus noise
- **ICA:** Find statistically independent (not just uncorrelated) components
- Key difference: PCA maximises variance, FA models noise, ICA maximises independence

163.1 Factor Analysis

Factor analysis assumes observed variables are linear combinations of latent factors plus noise. Unlike PCA, it explicitly models measurement noise.

Factor Analysis Model

Generative model:

$$x = Wz + \mu + \epsilon$$

where:

- $z \sim \mathcal{N}(0, I_L)$: latent factors (standard normal)
- $W \in \mathbb{R}^{D \times L}$: factor loadings (how factors influence observations)
- $\mu \in \mathbb{R}^D$: mean
- $\epsilon \sim \mathcal{N}(0, \Psi)$: noise with diagonal covariance $\Psi = \text{diag}(\psi_1, \dots, \psi_D)$

Reading the model: Each observed dimension x_d is a weighted combination of the latent factors (given by row d of W), plus a mean, plus noise specific to that dimension.

Implied distribution:

$$x \sim \mathcal{N}(\mu, WW^\top + \Psi)$$

Key difference from PCA: Factor analysis models **feature-specific noise** Ψ , while PCA treats all variance as signal. Each feature can have its own noise level ψ_d .

Comparison:

- **PCA:** Deterministic, finds directions of maximum variance, reconstructs via projection. Makes no distinction between signal and noise.
- **FA:** Probabilistic, models latent structure plus noise, fit via maximum likelihood or EM. Explicitly separates shared variance (from factors) and unique variance (noise).

When noise is homoscedastic ($\Psi = \sigma^2 I$), factor analysis reduces to probabilistic PCA (PPCA). Factor analysis is more appropriate when different features have different noise levels—common in psychometric applications (where different survey questions have different reliability).

163.2 Independent Component Analysis (ICA)

The Cocktail Party Problem

Scenario: L people speak simultaneously at a party. D microphones record mixtures of all voices. Can we recover the original voices?

Model:

$$x = As$$

where:

- $s \in \mathbb{R}^L$: source signals (the original voices)
- $A \in \mathbb{R}^{D \times L}$: mixing matrix (how sources combine at each microphone)
- $x \in \mathbb{R}^D$: observed mixtures (what the microphones record)

Goal: Recover s from x without knowing A . This seems impossible—we have one equation with two unknowns (A and s).

Key assumption: Source signals are **statistically independent** and **non-Gaussian**.

Why this helps: If we can find a transformation that makes the outputs maximally independent, we've found the original sources (up to scaling and ordering).

ICA vs PCA

PCA: Finds uncorrelated components (second-order statistics only)

$$\text{Cov}(z_i, z_j) = 0 \text{ for } i \neq j$$

Uncorrelated means: no *linear* relationship between components.

ICA: Finds independent components (all orders of statistics)

$$p(s_1, \dots, s_L) = \prod_{l=1}^L p(s_l)$$

Independent means: no relationship *whatever* between components—knowing one tells you nothing about the others.

Independence is strictly stronger than uncorrelatedness. For Gaussian distributions, uncorrelated implies independent (a special property of Gaussians), but not in general.

Example: Let $s_1 \sim \text{Uniform}(-1, 1)$ and $s_2 = s_1^2$. These are uncorrelated (covariance is zero) but clearly dependent (s_2 is determined by s_1).

Why non-Gaussianity matters: The Central Limit Theorem implies mixtures tend toward Gaussianity. When you add many independent random variables, the sum approaches a Gaussian. ICA exploits non-Gaussianity to identify the original (unmixed) sources. If sources were Gaussian, the problem would be unidentifiable—all rotations of Gaussian independent sources are also Gaussian and independent.

ICA algorithms (e.g., FastICA) typically maximise non-Gaussianity measures like kurtosis or negentropy. ICA is widely used in signal processing, neuroimaging (separating brain signals from

artifacts), and financial data analysis.

164 Manifold Learning

Section Summary: Manifold Learning

- **Manifold hypothesis:** High-dimensional data lies on/near a low-dimensional manifold
- **Why linear fails:** Manifolds can be curved; linear projections distort structure
- **t-SNE:** Preserve local neighbourhoods via probability matching
- **UMAP:** Topology-preserving, scalable, maintains more global structure
- **Use case:** Primarily visualisation; interpret with caution

164.1 The Manifold Hypothesis

The Manifold Hypothesis

Claim: Real-world high-dimensional data (images, text, audio) typically lies on or near a low-dimensional **manifold** embedded in the ambient space.

What is a manifold? A topological space that locally resembles Euclidean space. A d -dimensional manifold in \mathbb{R}^D can be locally parameterised by d coordinates.

Intuitive examples:

- A sphere is a 2D manifold in 3D space—locally it looks flat, but globally it curves
- A torus (doughnut shape) is also 2D in 3D
- The surface of the Earth is approximately a 2D sphere embedded in 3D

Data examples:

- Images of a rotating 3D object: 1D manifold (parameterised by angle). As the object rotates, the pixel values change continuously, tracing out a 1D curve in pixel space.
- Face images varying in pose, lighting, expression: low-dimensional manifold. A face image might have millions of pixels, but pose has 3 degrees of freedom, lighting maybe 5–10, expression maybe 10–20.
- Natural images: vast majority of 28×28 pixel arrays are “noise”—real digits form a tiny, structured subset of the $28^2 = 784$ -dimensional space.

Implication: The intrinsic dimensionality of the data is much lower than the ambient dimension, but the manifold may be highly curved and nonlinear.

164.2 Why Linear Methods Fail on Manifolds

Consider the “Swiss roll”—a 2D surface rolled up in 3D space. Points nearby on the roll surface might be far apart in 3D Euclidean distance (through the roll), and points far apart on the

surface might be close in 3D (on adjacent layers of the roll).

PCA would project onto a plane, collapsing points that are nearby in 3D but far apart along the manifold surface. Linear methods cannot “unroll” the manifold.

What we need: Methods that preserve **geodesic distances** (distances along the manifold) rather than Euclidean distances in ambient space.

164.3 Stochastic Neighbour Embedding (SNE)

SNE: Probability Model

Idea: Convert pairwise distances into probabilities representing “neighbour” relationships, then find a low-dimensional embedding that matches these probabilities.

High-dimensional similarities:

$$p_{j|i} = \frac{\exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma_i^2}\right)}{\sum_{k \neq i} \exp\left(-\frac{\|x_i - x_k\|^2}{2\sigma_i^2}\right)}$$

Reading this equation:

- $p_{j|i}$ represents the probability that x_i would choose x_j as a neighbour
- The numerator is a Gaussian kernel: nearby points ($\text{small } \|x_i - x_j\|$) get high values
- The denominator normalises so probabilities sum to 1 over all $j \neq i$
- σ_i controls how “local” point i ’s neighbourhood is (set adaptively per point)

This is essentially a softmax over negative squared distances.

Low-dimensional similarities:

$$q_{j|i} = \frac{\exp\left(-\|z_i - z_j\|^2\right)}{\sum_{k \neq i} \exp\left(-\|z_i - z_k\|^2\right)}$$

where z_i, z_j are low-dimensional representations (typically 2D for visualisation).

Objective: Minimise KL divergence between distributions:

$$\mathcal{L} = \sum_i D_{\text{KL}}(P_i \| Q_i) = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}$$

What this means: For each point i , we have two probability distributions over neighbours—one from high dimensions (P_i), one from the embedding (Q_i). We want these to match. KL divergence measures how different they are.

Perplexity: The bandwidth σ_i is set per-point to achieve a target perplexity (effective number of neighbours). Higher perplexity considers more neighbours, affecting global vs local structure.

Perplexity intuition: Perplexity $\approx 2^{\text{entropy}}$ of the neighbour distribution. A perplexity of 30 means each point effectively has about 30 neighbours. Low perplexity focuses on very local structure; high perplexity considers broader context.

164.4 t-Distributed SNE (t-SNE)

t-SNE: Heavy-Tailed Embedding

Problem with SNE: The Gaussian kernel in embedding space has light tails. This creates the “crowding problem”:

- In high dimensions, a point can have many “moderately distant” neighbours
- In 2D, there’s limited space—you can’t fit all those neighbours at moderate distances
- SNE forces moderately distant points too close together, crushing clusters

Solution: Use the Student’s t-distribution (1 degree of freedom = Cauchy) for low-dimensional similarities:

$$q_{ij} = \frac{(1 + \|z_i - z_j\|^2)^{-1}}{\sum_{k \neq l} (1 + \|z_k - z_l\|^2)^{-1}}$$

Why this helps: The t-distribution has **heavier tails** than the Gaussian. Points that are far apart in high dimensions can be modelled as *very* far apart in the embedding without excessive penalty. This creates better-separated clusters with more space between them.

t-SNE also symmetrises the high-dimensional probabilities:

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}$$

This ensures the relationship between i and j is symmetric: if i considers j a neighbour, j should also consider i a neighbour.

Gradient:

$$\frac{\partial \mathcal{L}}{\partial z_i} = 4 \sum_j (p_{ij} - q_{ij})(z_i - z_j)(1 + \|z_i - z_j\|^2)^{-1}$$

Interpreting the gradient:

- When $p_{ij} > q_{ij}$: points are closer in high-D than embedding \Rightarrow attractive force pulls them together
- When $p_{ij} < q_{ij}$: points are farther in high-D than embedding \Rightarrow repulsive force pushes them apart
- The $(1 + \|z_i - z_j\|^2)^{-1}$ factor comes from the t-distribution

NB!**t-SNE limitations and interpretation:**

- **$O(n^2)$ complexity:** Computing all pairwise interactions is expensive; approximations (Barnes-Hut) help but still limit scalability to tens of thousands of points
- **Non-parametric:** Cannot embed new points without re-running on full dataset. There's no learned function $f : \mathbb{R}^D \rightarrow \mathbb{R}^2$.
- **Perplexity sensitivity:** Results change significantly with perplexity; try multiple values (typically 5–50)
- **Global structure:** t-SNE prioritises local structure; distances *between* clusters are not meaningful. Two clusters appearing far apart may or may not be far in the original space.
- **Cluster sizes:** Visual cluster sizes don't reflect true cluster densities. A visually large cluster might have few points; a small tight cluster might have many.
- **Stochasticity:** Different runs give different results; use fixed random seeds for reproducibility. Consistent patterns across runs are more trustworthy.

t-SNE is primarily a **visualisation tool**—interpret cautiously and avoid drawing conclusions about global geometry.

164.5 UMAP

UMAP: Uniform Manifold Approximation and Projection

UMAP addresses t-SNE's limitations through a topological approach:

Key innovations:

1. ***k*-nearest neighbour graph:** Only consider local neighbourhoods, not all pairs. This reduces complexity from $O(n^2)$ to $O(n \log n)$ using efficient nearest neighbour algorithms.

2. **Local metric adaptation:** Each point has its own distance scale, handling varying densities:

$$d_i(x_i, x_j) = \max(0, \|x_i - x_j\| - \rho_i)/\sigma_i$$

where ρ_i is the distance to the nearest neighbour.

What this means: Distances are measured relative to each point's local density. In a dense region, “nearby” means very close; in a sparse region, “nearby” can mean farther away.

3. **Fuzzy simplicial sets:** Build a weighted graph representing local connectivity. Edges have weights in $[0, 1]$ representing strength of connection.

4. **Cross-entropy loss:** Optimise binary cross-entropy on edge existence:

$$\mathcal{L} = \sum_{i,j} \left[p_{ij} \log \frac{p_{ij}}{q_{ij}} + (1 - p_{ij}) \log \frac{1 - p_{ij}}{1 - q_{ij}} \right]$$

What this means: We're asking “does an edge exist between i and j ?” and optimising so the embedding's answer matches the high-dimensional answer. This is more symmetric than KL divergence.

Advantages over t-SNE:

- **Speed:** Near-linear scaling $O(n \log n)$ via approximate nearest neighbours
- **Global structure:** Preserves more large-scale relationships between clusters
- **Parametric option:** Can learn a mapping for embedding new points
- **Theoretical foundation:** Grounded in algebraic topology and Riemannian geometry

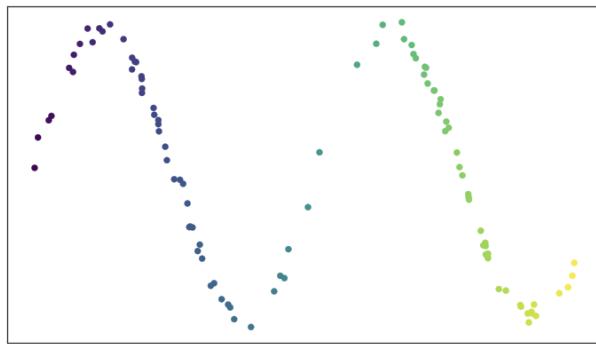


Figure 108: High-dimensional data projected to 2D, with colours indicating cluster membership. The goal of manifold learning is to find an embedding that reveals this structure.

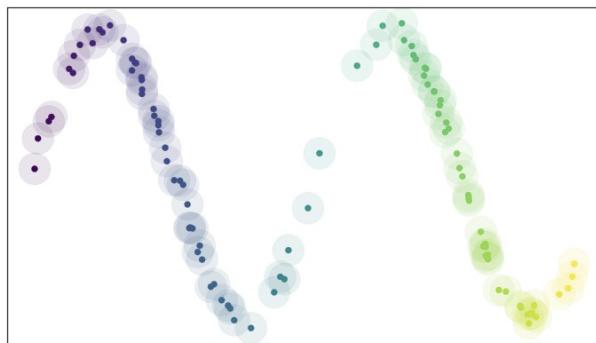


Figure 109: UMAP constructs a k -nearest neighbour graph; shaded circles represent local neighbourhoods. Each point connects to its k closest neighbours, capturing local structure efficiently.

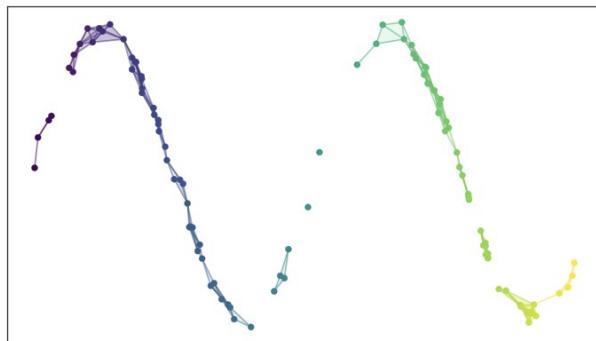


Figure 110: The embedding preserves neighbourhood structure: connected points in high dimensions remain close in the projection.

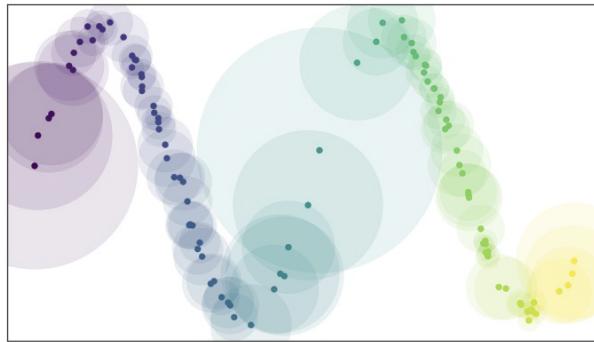


Figure 111: UMAP adapts to varying local densities, using different distance scales in dense vs sparse regions. The varying neighbourhood sizes (shaded regions) show this adaptation.

Choosing Between PCA, t-SNE, and UMAP

Method	Structure	Complexity	Use Case
PCA	Global, linear	$O(D^2n)$	Pre-processing, interpretable axes
t-SNE	Local, non-linear	$O(n^2)$	Visualisation (small data)
UMAP	Local + global	$O(n \log n)$	Visualisation (large data)

Guidelines:

- Start with PCA to understand linear structure and reduce noise
- Use t-SNE/UMAP for visualisation, not quantitative analysis
- UMAP for large datasets or when global structure matters
- Always try multiple hyperparameter settings

Interactive exploration: For building intuition about how t-SNE and UMAP behave with different hyperparameters, see <https://pair-code.github.io/understanding-umap/>

165 Clustering

Section Summary: Clustering

- **Goal:** Partition data into groups of similar points
- **K-means:** Minimise within-cluster variance; fast, assumes spherical clusters
- **Hierarchical:** Build tree of nested clusters; no need to prespecify K
- **DBSCAN:** Density-based; finds arbitrary shapes, handles noise
- **GMMs:** Soft clustering via mixture of Gaussians; probabilistic framework

Clustering partitions data into groups such that points within a group are more similar than points in different groups. Unlike classification, we have no labels—the algorithm must discover groupings from structure alone.

165.1 K-Means Clustering

K-Means Algorithm

Objective: Partition n points into K clusters $\{C_1, \dots, C_K\}$ minimising within-cluster sum of squares:

$$\mathcal{L} = \sum_{k=1}^K \sum_{x_i \in C_k} \|x_i - \mu_k\|^2$$

where $\mu_k = \frac{1}{|C_k|} \sum_{x_i \in C_k} x_i$ is the centroid of cluster k .

What this measures: The total squared distance from each point to its cluster centre. Low values mean points are close to their assigned centres.

Algorithm (Lloyd's algorithm):

1. **Initialise:** Choose K initial centroids μ_1, \dots, μ_K (e.g., random points, K-means++)
2. **Assign:** Assign each point to nearest centroid:

$$c_i = \arg \min_k \|x_i - \mu_k\|^2$$

What this does: Each point “votes” for the nearest centre.

3. **Update:** Recompute centroids:

$$\mu_k = \frac{1}{|C_k|} \sum_{i:c_i=k} x_i$$

What this does: Move each centre to the mean of its assigned points.

4. **Repeat** steps 2–3 until convergence (assignments don’t change)

Convergence: The objective decreases (or stays constant) at each step. Since there are finitely many partitions, the algorithm converges in finite time.

Why it works: The assign step minimises \mathcal{L} with respect to assignments (holding centres fixed). The update step minimises \mathcal{L} with respect to centres (holding assignments fixed). Alternating these can only decrease or maintain the objective.

NB!**K-means limitations:**

- **Assumes spherical clusters:** Poorly handles elongated or irregular shapes. K-means uses Euclidean distance, which is isotropic—it doesn't know about cluster shape.
- **Sensitive to initialisation:** Different starting points can give different results. Use multiple restarts (run 10–100 times, keep best) or K-means++ (smart initialisation that spreads initial centres).
- **Must specify K :** Number of clusters is a hyperparameter that must be chosen in advance.
- **Sensitive to outliers:** Outliers can pull centroids away from true cluster centres. A single outlier can dramatically shift a centroid.
- **Equal variance assumption:** Implicitly assumes clusters have similar sizes and variances. A tiny tight cluster and a large diffuse cluster will be treated equally.

165.2 Choosing K **Methods for Selecting K**

1. **Elbow method:** Plot within-cluster sum of squares vs K . Look for an “elbow” where the rate of decrease slows.

Intuition: Adding more clusters always reduces the objective (in the limit, $K = n$ gives zero error). But there's often a point of diminishing returns—the elbow.

2. **Silhouette score:** For each point i , compute:

$$s_i = \frac{b_i - a_i}{\max(a_i, b_i)}$$

where:

- a_i = average distance to points in same cluster (cohesion)
- b_i = average distance to points in nearest other cluster (separation)

$s_i \in [-1, 1]$; values near 1 indicate good clustering (tight within, separated between). Choose K maximising average silhouette.

3. **Gap statistic:** Compare within-cluster dispersion to that expected under a null reference distribution (uniform random data). Choose K where the gap is largest.

Intuition: How much better is our clustering than clustering random noise? The gap measures this improvement.

4. **Information criteria:** For probabilistic models (GMMs), use BIC or AIC. These penalise model complexity, balancing fit against number of parameters.

165.3 Hierarchical Clustering

Agglomerative Hierarchical Clustering

Idea: Build a hierarchy of clusters by iteratively merging the most similar clusters.

Algorithm:

1. Start with n clusters, one per point
2. Compute pairwise distances between all clusters
3. Merge the two closest clusters
4. Repeat until all points are in one cluster

Result: A **dendrogram**—a tree showing the order and distances of merges. Cut the tree at a chosen height to obtain a partition.

Dendrogram interpretation: The y-axis shows the distance at which clusters merge. Cutting at height h gives all clusters that exist at distance h . Higher cut = fewer, larger clusters.

Linkage criteria (how to measure distance between clusters A and B):

- **Single linkage:** $d(A, B) = \min_{a \in A, b \in B} d(a, b)$ — distance to nearest point
Effect: Can produce elongated “chaining” clusters. Two clusters might merge because of a single pair of close points, even if most points are far apart.
- **Complete linkage:** $d(A, B) = \max_{a \in A, b \in B} d(a, b)$ — distance to farthest point
Effect: Produces compact clusters. Forces all points to be within a certain distance.
- **Average linkage:** $d(A, B) = \frac{1}{|A||B|} \sum_{a,b} d(a, b)$ — average pairwise distance
Effect: Compromise between single and complete.
- **Ward’s method:** Minimise increase in total within-cluster variance
Effect: Tends to produce balanced clusters of similar size.

Advantages: No need to prespecify K ; produces interpretable hierarchy; works with any distance metric.

Disadvantages: $O(n^2)$ space and $O(n^3)$ time (or $O(n^2 \log n)$ with optimisations); cannot undo merges (greedy algorithm).

165.4 DBSCAN: Density-Based Clustering

DBSCAN Algorithm

Idea: Clusters are dense regions separated by sparse regions. Points in sparse regions are noise.

Parameters:

- ϵ : neighbourhood radius
- minPts: minimum points to form a dense region

Point classification:

- **Core point:** Has $\geq \text{minPts}$ points within distance ϵ
Intuition: In a “crowded” region with many neighbours.
- **Border point:** Within ϵ of a core point, but not itself core
Intuition: On the edge of a dense region.
- **Noise point:** Neither core nor border
Intuition: Isolated, not part of any cluster.

Algorithm:

1. Mark all points as core, border, or noise
2. Form clusters by connecting core points within ϵ of each other (density-reachable)
3. Assign border points to nearby clusters
4. Noise points remain unassigned

Advantages:

- Finds clusters of arbitrary shape (not just spheres)
- Automatically identifies noise/outliers
- Does not require specifying K

NB!**DBSCAN challenges:**

- **Parameter sensitivity:** Results depend heavily on ϵ and minPts. Wrong values can merge distinct clusters or fragment single clusters.
- **Varying densities:** Struggles when clusters have very different densities. A single ϵ can't be right for both a tight dense cluster and a loose sparse one.
- **High dimensions:** Density becomes less meaningful (curse of dimensionality). All points become approximately equidistant.

165.5 Gaussian Mixture Models**Gaussian Mixture Models (GMMs)**

Model: Data is generated from a mixture of K Gaussian distributions:

$$p(x) = \sum_{k=1}^K \pi_k \mathcal{N}(x | \mu_k, \Sigma_k)$$

where:

- π_k : mixing coefficient (probability of cluster k), with $\sum_k \pi_k = 1$
- μ_k : mean of cluster k
- Σ_k : covariance of cluster k (determines shape and orientation)

Generative story:

1. Choose cluster k with probability π_k
2. Sample x from $\mathcal{N}(\mu_k, \Sigma_k)$

Soft clustering: Each point has a probability of belonging to each cluster:

$$\gamma_{ik} = P(z_i = k | x_i) = \frac{\pi_k \mathcal{N}(x_i | \mu_k, \Sigma_k)}{\sum_j \pi_j \mathcal{N}(x_i | \mu_j, \Sigma_j)}$$

This is the **responsibility**—how much cluster k is responsible for point i .

Reading this equation: It's Bayes' theorem. The numerator is the prior (π_k) times likelihood (Gaussian density); the denominator normalises.

Relation to K-means: K-means is a special case where all covariances are $\sigma^2 I$ (spherical) and we take the limit $\sigma^2 \rightarrow 0$ (hard assignment). In this limit, responsibilities become binary (0 or 1), and we recover K-means.

EM Algorithm for GMMs

The EM algorithm iteratively maximises the likelihood:

E-step (compute responsibilities):

$$\gamma_{ik} = \frac{\pi_k \mathcal{N}(x_i | \mu_k, \Sigma_k)}{\sum_j \pi_j \mathcal{N}(x_i | \mu_j, \Sigma_j)}$$

What this does: Given current parameters, compute how much each cluster “claims” each point.

M-step (update parameters):

$$\begin{aligned} N_k &= \sum_{i=1}^n \gamma_{ik} \quad (\text{effective number of points in cluster } k) \\ \mu_k &= \frac{1}{N_k} \sum_{i=1}^n \gamma_{ik} x_i \quad (\text{weighted mean}) \\ \Sigma_k &= \frac{1}{N_k} \sum_{i=1}^n \gamma_{ik} (x_i - \mu_k)(x_i - \mu_k)^\top \quad (\text{weighted covariance}) \\ \pi_k &= \frac{N_k}{n} \quad (\text{fraction of points in cluster } k) \end{aligned}$$

What this does: Given responsibilities, find the best parameters. Each point contributes to each cluster’s statistics, weighted by responsibility.

Convergence: EM is guaranteed to increase the log-likelihood at each step (or leave it unchanged), converging to a local maximum.

Clustering Method Comparison

Method	Shape	K needed?	Noise?	Scalability
K-means	Spherical	Yes	No	$O(nKt)$
Hierarchical	Any	No	No	$O(n^2)$ – $O(n^3)$
DBSCAN	Arbitrary	No	Yes	$O(n \log n)$
GMM	Elliptical	Yes	No	$O(nK^2D^2t)$

When to use what:

- **K-means:** Quick exploration, roughly spherical clusters, large datasets
- **Hierarchical:** When you want to explore different granularities, small datasets
- **DBSCAN:** Irregular shapes, when noise/outliers are expected
- **GMM:** When you need soft assignments, elliptical clusters, probabilistic interpretation

166 Autoencoders

Section Summary: Autoencoders

- **Architecture:** Encoder compresses to bottleneck; decoder reconstructs
- **Training:** Self-supervised—input is its own target
- **Key insight:** Linear autoencoder = PCA; nonlinear captures manifold structure
- **Variants:** Denoising (robustness), sparse (interpretability), variational (generation)

Autoencoders learn nonlinear dimensionality reduction using neural networks. The key idea is **self-supervision**: reconstruct the input from a compressed intermediate representation. No labels are needed—the input is its own target.

166.1 Architecture and Training

Autoencoder Architecture

An autoencoder consists of:

1. **Encoder** $f_\phi : \mathbb{R}^D \rightarrow \mathbb{R}^L$ compresses input to latent space

Typically a neural network with decreasing layer sizes: $D \rightarrow 512 \rightarrow 256 \rightarrow L$

2. **Bottleneck** layer with $L < D$ dimensions (the learned representation)

This is where the compression happens. The encoder must distil the input into L numbers.

3. **Decoder** $g_\psi : \mathbb{R}^L \rightarrow \mathbb{R}^D$ reconstructs input from latent code

Typically mirrors the encoder: $L \rightarrow 256 \rightarrow 512 \rightarrow D$

Objective:

$$\hat{\phi}, \hat{\psi} = \arg \min_{\phi, \psi} \frac{1}{n} \sum_{i=1}^n \|x_i - g_\psi(f_\phi(x_i))\|^2$$

Breaking this down:

- $f_\phi(x_i) = z_i$: encode input to latent representation
- $g_\psi(z_i) = \hat{x}_i$: decode latent back to reconstruction
- $\|x_i - \hat{x}_i\|^2$: reconstruction error (squared L2 distance)
- We minimise average reconstruction error over the dataset

The bottleneck forces the network to learn a compressed representation capturing the most important features.

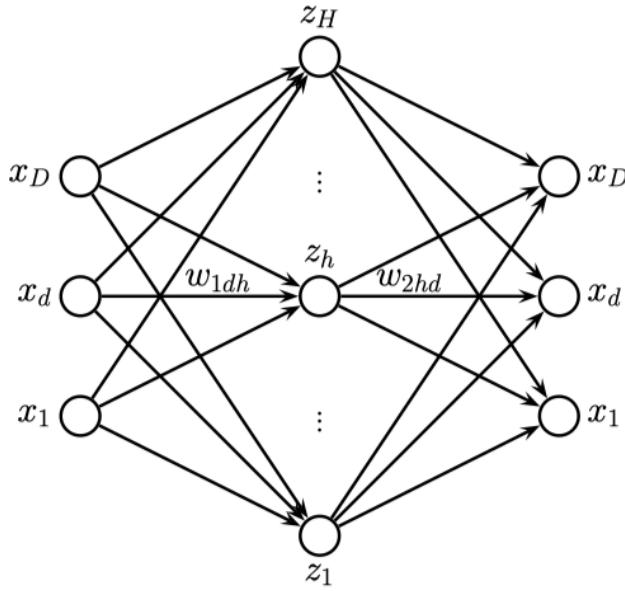


Figure 112: Autoencoder architecture: input is compressed through the encoder to a bottleneck layer, then reconstructed by the decoder. The bottleneck dimension L is smaller than input dimension D , forcing compression.

Training: Standard backpropagation through the entire network. The loss gradient flows from reconstruction error back through decoder and encoder. Both networks learn jointly.

166.2 Relationship to PCA

Linear Autoencoders Recover PCA

Under specific conditions, autoencoders are equivalent to PCA:

- Single hidden layer
- Linear activations (no nonlinearities)
- Mean squared error loss

Proof sketch: With linear encoder $z = W_{\text{enc}}^\top x$ and decoder $\hat{x} = W_{\text{dec}}z$, the objective is:

$$\min_{W_{\text{enc}}, W_{\text{dec}}} \|X - XW_{\text{enc}}W_{\text{dec}}^\top\|_F^2$$

The optimal solution has $W_{\text{dec}} = W_{\text{enc}}$ and W_{enc} spans the same subspace as the top L principal components. The columns of W_{enc} need not be the eigenvectors exactly (any rotation within the subspace works), but the span is identical.

Why this matters: It shows PCA is a special case of autoencoders. The autoencoder framework is strictly more general—adding nonlinearities lets us capture manifold structure that PCA cannot.

With nonlinear activations and multiple layers, autoencoders can learn **nonlinear manifolds** that PCA cannot capture.

166.3 Bottleneck Dimension and Regularisation

NB!

Avoiding trivial solutions:

- If $L \geq D$, the autoencoder can learn the identity function—just copy the input through without compression
- Even with $L < D$, an overly expressive network might memorise rather than generalise
- The bottleneck must constrain capacity to force meaningful compression

Regularisation strategies:

- Use smaller bottleneck dimension
- Add weight decay to encoder/decoder weights
- Use dropout during training
- Add noise (denoising autoencoder)
- Impose sparsity on latent activations

166.4 Autoencoder Variants

Denoising Autoencoders

Idea: Corrupt inputs and train to reconstruct the clean version.

Training:

1. Corrupt input: $\tilde{x}_i = x_i + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$ (or masking, dropout)
2. Train to reconstruct clean input: $\min \|x_i - g_\psi(f_\phi(\tilde{x}_i))\|^2$

Note: The target is the *clean* input x_i , not the corrupted input \tilde{x}_i .

Benefits:

- Forces robust representations invariant to small perturbations
- Prevents identity learning even with large bottleneck (can't memorise random noise)
- Learns to denoise as a side effect

Interpretation: The autoencoder learns to project noisy inputs onto the data manifold. It learns what “clean” data looks like.

Sparse Autoencoders

Idea: Encourage most latent units to be inactive for any given input.

Objective:

$$\mathcal{L} = \underbrace{\frac{1}{n} \sum_i \|x_i - \hat{x}_i\|^2}_{\text{reconstruction}} + \lambda \underbrace{\sum_i \|z_i\|_1}_{\text{sparsity}}$$

The L1 penalty $\|z_i\|_1 = \sum_l |z_{il}|$ encourages sparsity—many components being exactly zero.

Benefits:

- Encourages disentangled representations (each unit captures distinct feature)
- Can use overcomplete representations ($L > D$) without learning identity
- Analogous to L1 regularisation in linear models (Lasso)

Intuition: Different inputs activate different subsets of latent units. This can lead to more interpretable representations where each latent dimension means something specific.

166.5 Variational Autoencoders

VAEs combine autoencoders with probabilistic modelling, enabling both representation learning and generation.

VAE: Generative Model

Generative story:

1. Sample latent code: $z \sim p(z) = \mathcal{N}(0, I)$
2. Generate observation: $x \sim p_\theta(x | z)$

What this describes: A generative process for creating data. First sample a random “recipe” z from a simple distribution, then use the decoder to generate an observation.

Goal: Learn θ (decoder parameters) to maximise $\log p_\theta(x)$.

Problem: The marginal likelihood $p_\theta(x) = \int p_\theta(x | z)p(z)dz$ is intractable—we can’t compute this integral analytically.

Solution: Introduce an approximate posterior $q_\phi(z | x)$ (the encoder) and optimise a lower bound.

Evidence Lower Bound (ELBO)

For any distribution $q_\phi(z | x)$:

$$\begin{aligned}\log p_\theta(x) &= \log \int p_\theta(x | z)p(z)dz \\ &= \log \int q_\phi(z | x) \frac{p_\theta(x | z)p(z)}{q_\phi(z | x)} dz \\ &\geq \int q_\phi(z | x) \log \frac{p_\theta(x | z)p(z)}{q_\phi(z | x)} dz \quad (\text{Jensen's inequality})\end{aligned}$$

Step-by-step:

- Line 1: The quantity we want to maximise
- Line 2: Multiply and divide by $q_\phi(z | x)$ inside the integral
- Line 3: Apply Jensen's inequality: $\log \mathbb{E}[X] \geq \mathbb{E}[\log X]$ for concave \log

Rearranging:

$$\log p_\theta(x) \geq \underbrace{\mathbb{E}_{q_\phi(z|x)} [\log p_\theta(x | z)]}_{\text{reconstruction term}} - \underbrace{D_{\text{KL}}(q_\phi(z | x) \| p(z))}_{\text{regularisation term}}$$

This is the **Evidence Lower Bound (ELBO)**.

Interpretation:

- **Reconstruction:** Encode x to z , decode back; should recover x . This is the expected log-likelihood of the data given the latent code.
- **Regularisation:** Keep the approximate posterior $q_\phi(z | x)$ close to the prior $p(z) = \mathcal{N}(0, I)$. This prevents the encoder from learning arbitrary, unstructured latent codes.

VAE Architecture and Training

Encoder outputs distribution parameters:

$$q_\phi(z | x) = \mathcal{N}(\mu_\phi(x), \text{diag}(\sigma_\phi^2(x)))$$

A neural network takes x and outputs vectors μ and $\log \sigma^2$.

Reparameterisation trick: To backpropagate through sampling:

$$z = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

Why this is needed: We can't backpropagate through a random sample. But we can backpropagate through a deterministic function of the parameters plus external randomness. The trick rewrites sampling as a deterministic transformation of μ , σ , and noise ϵ .

Decoder outputs reconstruction distribution:

$$p_\theta(x | z) = \mathcal{N}(x | \mu_\theta(z), \sigma^2 I)$$

(For images, often use Bernoulli with cross-entropy loss.)

Loss function:

$$\mathcal{L} = -\mathbb{E}_\epsilon[\log p_\theta(x | z)] + D_{\text{KL}}(q_\phi(z | x) \| \mathcal{N}(0, I))$$

The KL term has closed form for Gaussians:

$$D_{\text{KL}} = \frac{1}{2} \sum_{j=1}^L (\mu_j^2 + \sigma_j^2 - \log \sigma_j^2 - 1)$$

Practical training: Sample one ϵ per data point, compute loss, backpropagate. The Monte Carlo estimate has low variance because the reparameterisation makes gradients well-behaved.

VAE: Key Properties

- **Generation:** Sample $z \sim \mathcal{N}(0, I)$, decode to get new data. Because the prior is simple, sampling is easy.
- **Interpolation:** Interpolate in latent space between two points; decode to see smooth transitions. The regularised latent space makes this work.
- **Regularised latent space:** KL term encourages smooth, organised latent space where nearby points decode to similar outputs.
- **Trade-off:** Stronger KL penalty \Rightarrow smoother latent space but blurrier reconstructions (the “posterior collapse” problem in extreme cases, where the encoder ignores the input and the decoder learns to generate from the prior alone).

When to Use Each Autoencoder Variant

- **Standard autoencoder:** Feature learning, pre-training, anomaly detection (high reconstruction error indicates anomaly)
- **Denoising:** Robust features, when data is noisy, preventing identity shortcuts
- **Sparse:** Interpretable, disentangled representations
- **Variational:** Generative modelling, sampling new data, smooth latent space

167 Self-Supervised Learning

Section Summary: Self-Supervised Learning

- **Idea:** Create supervision signal from data itself (no manual labels)
- **Pretext tasks:** Predict transformations, rotations, masked content
- **Contrastive learning:** Pull similar pairs together, push dissimilar pairs apart
- **Goal:** Learn representations useful for downstream tasks

Self-supervised learning occupies the space between supervised and unsupervised learning. It uses the structure of unlabelled data to create pseudo-labels, then trains with standard supervised objectives. This has become the dominant paradigm for pre-training large models.

Key insight: We can create labels automatically from the data's structure. Predict the next word in a sentence (language models). Predict whether two image patches came from the same image. Predict the rotation applied to an image. None of these requires human annotation.

167.1 Pretext Tasks

Pretext Tasks for Representation Learning

Idea: Design a task where labels can be generated automatically from the data. Solving this task requires learning useful representations.

Examples:

- **Rotation prediction:** Rotate images by $\{0^\circ, 90^\circ, 180^\circ, 270^\circ\}$; predict the rotation.

Why it works: To predict rotation, the network must understand object orientation—what's “up” vs “down”. This requires understanding object structure.

- **Jigsaw puzzles:** Divide image into patches, shuffle; predict the original arrangement.

Why it works: Requires understanding spatial relationships—what typically appears next to what.

- **Colourisation:** Given grayscale image, predict colours.

Why it works: Requires understanding object semantics—grass is green, sky is blue, skin is skin-toned.

- **Masked prediction:** Mask part of input (e.g., words in text, patches in images); predict the masked content.

Why it works: Requires understanding context—what words/features typically occur in this context? This is the foundation of BERT and masked autoencoders.

- **Context prediction:** Given a patch, predict which of several candidate patches was its neighbour.

Why it works: Requires understanding spatial context and co-occurrence patterns.

Key insight: The pretext task itself is not the goal. We care about the representations learned in the process, which transfer to downstream tasks.

167.2 Contrastive Learning

Contrastive Learning Framework

Core idea: Learn representations where similar items are close and dissimilar items are far apart.

Setup:

- Create **positive pairs**: Two views of the same data point (e.g., different augmentations of the same image—crop, flip, colour jitter)
- Create **negative pairs**: Views from different data points

Objective: Learn an encoder f such that:

$$\text{sim}(f(x), f(x^+)) \gg \text{sim}(f(x), f(x^-))$$

where x^+ is a positive pair with x and x^- is a negative.

Similarity is typically cosine similarity: $\text{sim}(u, v) = u^\top v / (\|u\| \|v\|)$

InfoNCE loss (used in SimCLR, MoCo, etc.):

$$\mathcal{L} = -\log \frac{\exp(\text{sim}(z_i, z_j)/\tau)}{\sum_{k \neq i} \exp(\text{sim}(z_i, z_k)/\tau)}$$

where:

- z_i, z_j are representations of positive pair (same image, different augmentations)
- τ is temperature (controls sharpness—lower temperature makes the loss focus more on hard negatives)
- Denominator includes all other examples in the batch as negatives

Reading this loss: It's a softmax classification loss. We want to classify the positive pair correctly among all negatives. Minimising this loss pushes the positive pair together and all negatives apart.

SimCLR: A Simple Framework for Contrastive Learning

Architecture:

1. **Data augmentation:** Apply random transformations to create two views of each image (crop, flip, colour jitter, blur)
2. **Encoder:** CNN (e.g., ResNet) extracts representations $h = f(x)$
3. **Projection head:** MLP maps to space where contrastive loss is applied: $z = g(h)$
4. **Contrastive loss:** InfoNCE treating other images in batch as negatives

Key findings:

- Strong augmentations are crucial (composition of multiple transforms). The harder the augmentation, the more the network must learn semantic features to recognise same-image pairs.
- Larger batch sizes help (more negatives). With 4096 batch size, each positive has 8190 negatives.
- The projection head is essential during training but discarded for downstream tasks. The pre-projection representation h transfers better.
- Learned representations transfer well to supervised tasks with limited labels.

167.3 Non-Contrastive Methods

Beyond Contrastive Learning

Recent methods avoid explicit negative pairs, addressing challenges like needing large batch sizes:

BYOL (Bootstrap Your Own Latent):

- Two networks: online and target (momentum-updated, i.e., slowly-moving average of online)
- Online network predicts target network's representation
- Asymmetry prevents collapse (both networks outputting constant)

Why it doesn't collapse: The target network changes slowly (momentum update), so the online network must keep improving to predict it. The asymmetry breaks the trivial equilibrium.

SimSiam:

- Siamese network with stop-gradient on one branch
- No negatives, no momentum encoder
- Stop-gradient is crucial for preventing collapse

Key insight: Stop-gradient prevents the network from learning a trivial constant solution.

Masked Autoencoders (MAE):

- Mask large portion (75%) of image patches
- Train to reconstruct masked patches
- Works well for Vision Transformers

Key insight: With so much masked, the task is genuinely hard. The network must understand image structure to fill in missing regions.

Self-Supervised Learning: Key Takeaways

- Self-supervision enables learning from vast unlabelled data
- Learned representations often rival or exceed supervised pre-training
- Data augmentation encodes inductive biases about what should be invariant
- Foundation for modern large-scale models (GPT, BERT, CLIP)
- Connection to unsupervised learning: both exploit data structure without labels

168 Summary and Connections

Chapter Summary: Unsupervised Learning

Dimensionality Reduction:

- PCA: Linear projection maximising variance; optimal for Gaussian data
- Factor Analysis: Adds feature-specific noise modelling
- ICA: Finds independent (not just uncorrelated) components
- Kernel PCA: Nonlinear extension via the kernel trick

Manifold Learning:

- Manifold hypothesis: Data lies on low-dimensional manifold
- t-SNE: Preserves local neighbourhoods; visualisation tool
- UMAP: Scalable, preserves more global structure

Clustering:

- K-means: Fast, assumes spherical clusters
- Hierarchical: Produces dendrogram, no fixed K
- DBSCAN: Density-based, handles noise
- GMMs: Probabilistic soft clustering

Representation Learning:

- Autoencoders: Neural compression; linear \equiv PCA
- VAEs: Probabilistic + generative; regularised latent space
- Self-supervised: Pretext tasks and contrastive learning

Connections Across the Course

- **Linear autoencoder \equiv PCA:** Constraint on architecture determines what structure is learned
- **VAEs and Bayesian methods:** ELBO connects to variational inference (Week 9: Uncertainty)
- **Kernel PCA and kernel methods:** Same kernel trick as kernel ridge regression (Week 6: Kernels)
- **GMMs and EM:** Iterative optimisation like coordinate descent; local optima issues
- **Self-supervised pre-training:** Improves supervised learning with limited labels
- **Embeddings in neural networks:** Hidden layers learn representations (Weeks 10–11: Neural Networks)

Terminology: Embeddings Revisited

An **embedding** is a mapping from high-dimensional or discrete data to a lower-dimensional continuous space that preserves relevant structure.

Embeddings appear throughout machine learning:

- Word embeddings (Word2Vec, GloVe): words \rightarrow dense vectors capturing semantic relationships (“king” - “man” + “woman” \approx “queen”)
- Graph embeddings: nodes \rightarrow vectors preserving graph structure
- Image embeddings: images \rightarrow feature vectors from CNN intermediate layers
- PCA: Linear embedding maximising variance
- t-SNE/UMAP: Nonlinear embedding preserving local structure
- Autoencoders: Learned nonlinear embedding via reconstruction
- Contrastive learning: Embedding where similar items are close

The term “embedding” emphasises that we’re not just reducing dimensions but *embedding* data into a space where geometric relationships carry semantic meaning.

Practical Guidelines

When to use what:

- **PCA**: First step in exploration; interpretable, fast, removes noise
- **t-SNE/UMAP**: Visualisation of clusters in high-dimensional data
- **K-means**: Quick clustering when clusters are spherical
- **DBSCAN**: Clusters of arbitrary shape; noise handling needed
- **GMM**: Soft clustering; elliptical clusters; probabilistic interpretation needed
- **Autoencoders**: Nonlinear dimensionality reduction; pre-training for deep learning
- **VAEs**: Generation; interpolation in latent space
- **Contrastive learning**: Pre-training with large unlabelled datasets

Common workflow:

1. Standardise features (if not comparable scales)
2. PCA for initial exploration and noise reduction
3. t-SNE/UMAP for visualisation
4. Clustering to identify groups
5. Validate clusters with domain knowledge