Intellectual History of AI and Machine 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 $\Gamma$
- 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 $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$

The machine operates deterministically: given current state $q \in Q$ and symbol $s \in \Gamma$ under the head, the transition function specifies a new symbol to write, a direction to move, and a new state to enter. 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 $\epsilon$.

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

**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 $x_1, \ldots, x_n$, weights $w_1, \ldots, w_n$, and threshold $\theta$ computes:

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

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.

**Limitations**: The model was highly idealised. Real neurons have continuous-valued outputs, complex temporal dynamics, 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 Rule**

In modern notation, the Hebbian update for the weight $w_{ij}$ from neuron $i$ to neuron $j$ is:

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

where $\eta > 0$ is a learning rate, $x_i$ is the presynaptic activity, and $y_j$ is the postsynaptic activity. The weight increases when both neurons are active simultaneously.
**Problem**: Pure Hebbian learning has no mechanism for weights to decrease, leading to unbounded growth. Modern variants (Oja's rule, BCM theory) incorporate normalisation or competition to address this.

## 3 The Birth of Artificial Intelligence (1956)

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.
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
- **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)
- 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

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 $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ 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 $f : \mathcal{X} \to \mathcal{Y}$ that generalises well to unseen data from the same distribution.
**Classification**: $\mathcal{Y}$ is discrete (e.g., $\{0, 1\}$ for binary classification, or $\{1, \ldots, K\}$ 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:

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

where $L$ is a loss function measuring prediction error and $\mathcal{F}$ 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, and $\text{sign}(z) = +1$ if $z \geq 0$, else $-1$.
**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. The update pushes the decision boundary in a direction that would correctly classify the current example.
**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. The number of mistakes is bounded by $(R/\gamma)^2$ where $R$ is the radius of the data and $\gamma$ is the margin of the best separating hyperplane.
**Intuition**: Each update rotates the weight vector toward correctly classifying the current mistake. If a perfect separator exists, we eventually find it.

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)

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.

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)

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:

$$\text{IF } \langle \text{condition} \rangle \text{ THEN } \langle \text{action} \rangle$$

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 could proceed in two 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
- **Backward chaining** (goal-driven): Start from the goal, find rules whose conclusions match, then try to establish those rules' premises (recursively)

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)

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 $\theta$ (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 $l$:

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

where $a_j^{(l)} = \sum_i w_{ij}^{(l)} h_i^{(l-1)}$ is the pre-activation, $h_i^{(l-1)}$ is the activation of unit $i$ in the previous layer, and $\delta_j^{(l)} = \frac{\partial L}{\partial a_j^{(l)}}$ is the "error signal" at unit $j$.
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 $\sigma'$ is the derivative of the activation function. Errors "back-propagate" from output to input.
This reduces the complexity from $O(W^2)$ (naive numerical differentiation, perturbing each weight) to $O(W)$ where $W$ is the total number of weights—a massive speedup that makes training practical.

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)

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

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 $\mathcal{D} = \{x_1, \ldots, x_n\}$ 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.
Common tasks include:

- **Clustering**: Partition data into groups of similar points ($k$-means, hierarchical clustering, DBSCAN). Each point is assigned to a cluster $z_i \in \{1, \ldots, K\}$.
- **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).

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

- **Hopfield Networks** (1982): Recurrent networks that function as associative (content-addressable) memory, using an energy function framework from statistical physics
- **Boltzmann Machines** (Hinton & Sejnowski, 1983): Stochastic neural networks that learn probability distributions over their inputs, providing a principled probabilistic framework for neural computation

# 10 Reinforcement Learning

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

The goal is to learn a **policy** $\pi : \mathcal{S} \to \mathcal{A}$ (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 $\pi$:

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

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

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

This recursive equation says: the optimal value of a state is the best action's immediate reward plus the discounted optimal value of the resulting state. Dynamic programming algorithms (value iteration, policy iteration) solve this when the MDP is known.

Key developments in RL:

- **Temporal Difference Learning** (Sutton, 1988): Learning value functions from experience without waiting for episode completion. TD learning bootstraps—it updates estimates based on other estimates.
- **Q-Learning** (Watkins, 1989): A model-free algorithm for learning optimal policies without knowing transition probabilities. Learns the Q-function $Q(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s')$.

- **TD-Gammon** (Tesauro, 1992): An RL system that achieved expert-level backgammon play through self-play, learning entirely from the game outcomes. A landmark demonstration of RL's potential.

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

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

## 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)

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
- **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
- **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. The computational performance gains for matrix multiplication were transformative.

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.

3. **Algorithms**: Better architectures and training techniques solved old problems:

   - ReLU activations (instead of sigmoid) reduced vanishing gradients
   - Dropout provided effective regularisation
   - Batch normalisation stabilised training
   - Residual connections enabled 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 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. 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:

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

**Interpretation**: $QK^\top$ computes similarity scores between each query and all keys. Dividing by $\sqrt{d_k}$ prevents scores from becoming too large (which would make softmax saturate). The softmax converts scores to attention weights (probabilities). Finally, we take a weighted sum of values.
**Multi-head attention** runs several attention operations in parallel with different learned projections, allowing the model to attend to different types of relationships simultaneously.

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 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

**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.

# ML Lecture Notes: Week 2
# Training, Divergence, Loss & Optimisation

## 14 The Optimisation 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

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)

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

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

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

The quantity $p(\mathcal{D}|\theta)$ is called the **likelihood**. Note the perspective here: we treat the data $\mathcal{D}$ as fixed (it's what we observed) and ask which parameter values $\theta$ would have made this data most probable. This is the **frequentist** perspective—parameters are fixed but unknown constants, and data is the random variable.

**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.
2. **Identical distribution**: All observations follow the same model with the same parameters.

**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.).

## 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
2. **Mathematical awkwardness**: Derivatives of products are messy (product rule applied repeatedly)

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:

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

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

**Why Negative Log-Likelihood?**

1. **Numerical stability**: Sums don't underflow like products of small numbers
2. **Computational convenience**: Derivatives of sums are sums of derivatives
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

# 16 Linear Regression as MLE

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 $\sigma^2$.

**Linear Regression Model**

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

Model parameters: $\theta = (\beta, \sigma^2)$
The probability density for observation $y_i$:

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

Let's unpack this model:

- **Simple Normal Model**: If $y_i \sim \mathcal{N}(\mu, \sigma)$ with fixed mean, the parameters are $\theta = (\mu, \sigma)$. All observations come from a normal distribution with the same mean and variance.
- **Normal Model with Linear Predictor**: If $y_i \sim \mathcal{N}(\mu_i, \sigma)$ where $\mu_i = x_i^\top \beta$, then $\theta = (\beta, \sigma)$. The mean varies across observations as a linear function of predictors—this is **linear regression**.

**Linear Regression**

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 $\sigma^2$ is constant across all observations

This forms the basis of linear regression models, where we assume normally distributed errors.

## 16.2 Deriving the NLL

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

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

Summing over all observations:

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

**MLE**

The NLL has two terms:

1. $\frac{n}{2}\log(2\pi\sigma^2)$: Depends only on $\sigma^2$ (constant w.r.t. $\beta$)
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 $\beta$ is equivalent to minimising the **Residual Sum of Squares (RSS)**. The Gaussian assumption justifies least squares!

**The MLE-RSS Connection**

The goal in MLE is to find the parameter values that minimise the NLL. For linear regression with Gaussian errors:

- The first term $\frac{n}{2}\log(2\pi\sigma^2)$ is constant for a given $\sigma$ and affects all observations equally
- The second term $\frac{1}{2\sigma^2}\sum_i(y_i - x_i^\top\beta)^2$ penalises deviations of observed values from their predicted means
- Since $\sigma^2$ is a positive constant (w.r.t. $\beta$), minimising NLL over $\beta$ is equivalent to minimising the sum of squared residuals

This means MLE for linear regression with Gaussian errors produces the same $\hat{\beta}$ as ordinary least squares (OLS).

# 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**: $\text{MSE}(\beta) = \frac{1}{n}\text{RSS}(\beta)$

The RSS measures the total squared deviation between observed values and predictions. The factor of $\frac{1}{2}$ sometimes appears for convenience (it cancels with the 2 from differentiation), but doesn't affect the optimal $\beta$.

## 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 $\beta$—it has a unique global minimum.

**Derivation of OLS Estimator**

**Step 1**: Expand RSS

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

(The middle terms are equal since they're scalars and $(y^\top X\beta)^\top = \beta^\top X^\top y$.)
**Step 2**: Take the gradient w.r.t. $\beta$
Using matrix calculus identities $\nabla_\beta(\beta^\top a) = a$ and $\nabla_\beta(\beta^\top A\beta) = 2A\beta$ for symmetric $A$:

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

**Step 3**: Set to zero and solve

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

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

**NB!**

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

- $n < p$ (more features than observations)—the system is underdetermined
- Features are **perfectly collinear**—one feature is an exact linear combination of others
- Features are **near-collinear**—numerical instability, huge variance in estimates

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

## 17.2 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 (or new data with the same features)
- **Interpret**: $\hat{\beta}_j$ is the expected change in $y$ for a unit change in $x_j$, *holding other features constant*. More precisely:
$$\frac{\partial \hat{y}}{\partial x_j} = \hat{\beta}_j$$
- **Quantify uncertainty**: Via $\text{Var}(\hat{\beta})$—how much would our estimates change with different data?

# 18 KL Divergence and Cross-Entropy

An alternative motivation for MLE comes from information theory: we want a model distribution $q_\theta$ that is "close" to the true data distribution $p$. But what does "close" mean for probability distributions?

## 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 $p$:

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

This can be decomposed as:

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

So: $D_{\mathrm{KL}}(p\|q) = H(p,q) - H(p)$
Properties:

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

The asymmetry matters! $D_{\mathrm{KL}}(p\|q)$ penalises cases where $p(y) > 0$ but $q(y) \approx 0$ (the model assigns low probability to events that actually happen). This is appropriate for density estimation.

**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:

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

In practice, we don't know $p$, but we can estimate the cross-entropy from samples:

$$H(p, q_\theta) \approx -\frac{1}{n}\sum_{i=1}^{n} \log q_\theta(y_i) = \frac{1}{n}\mathrm{NLL}(\theta)$$

Thus: **MLE finds the model closest to the data in the KL sense.**

# 19 Variance of the OLS Estimator

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 shows that $\hat{\beta}$ equals the true $\beta$ plus a random perturbation that depends on the errors $\epsilon$.
**Unbiasedness**: Since $\mathbb{E}[\epsilon] = 0$:

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

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

**Sandwich Form of Variance**

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

This is the **sandwich estimator**:

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

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:

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

The variance $\sigma^2$ is estimated by:

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

(We divide by $n - p$ rather than $n$ for unbiasedness—we "used up" $p$ degrees of freedom estimating $\beta$.)
The **standard error** of $\hat{\beta}_j$ is: $\mathrm{SE}(\hat{\beta}_j) = \hat{\sigma}\sqrt{[(X^\top X)^{-1}]_{jj}}$

## 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:

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

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 $\text{diag}(\hat{e}_1^2, \ldots, \hat{e}_n^2)$
3. Compute the sandwich: $(X^\top X)^{-1} X^\top \text{diag}(\hat{e}^2) X (X^\top X)^{-1}$
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

MLE treats parameters as fixed unknowns. **Bayesian inference** treats parameters as random variables with distributions reflecting uncertainty.

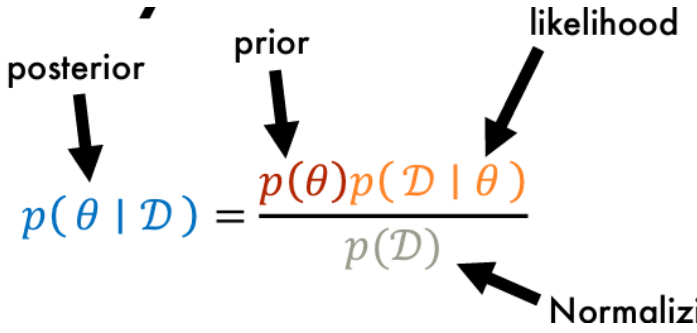



Figure 1: Bayes' Rule: combining prior beliefs with observed data to form posterior beliefs. The posterior balances what we believed before (prior) with what the data tells us (likelihood).

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

## 20.1 Frequentist vs Bayesian Perspectives

**Two Philosophies of Probability**

| | **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?"

## 20.2 Maximum A Posteriori (MAP) Estimation

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

**MAP Estimation**

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

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

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

Taking logs (for computational convenience):

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

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

**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)
- **Laplace prior**: $p(\theta) \propto \exp(-\lambda\|\theta\|_1) \Rightarrow$ **Lasso** (L1 penalty)
- **Uniform/flat prior** (improper): $p(\theta) \propto 1 \Rightarrow$ MAP = MLE

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

### 20.3 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 $\beta$ 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 $\beta$ is fixed, data is random
- "If we repeated this experiment many times, 95% of the constructed intervals would contain the true $\beta$"
- Statement about the *procedure*, not the parameter
- We don't assume a distribution for $\beta$; 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 $\sigma^2$, use $t$-distribution critical values instead:

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

# 21 Empirical Risk Minimisation

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**: $\ell(y, \hat{y}) = \mathbf{1}[y \neq \hat{y}]$

## 21.1 Common Loss Functions

| Loss | Formula | Use Case |
|---|---|---|
| Squared (L2) | $(y - \hat{y})^2$ | Regression |
| Absolute (L1) | $\lvert y - \hat{y} \rvert$ | 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 |

## 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
- **Non-differentiable**: Can't use gradient-based optimisation
- **Flat almost everywhere**: Gradient is zero except at decision boundary

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

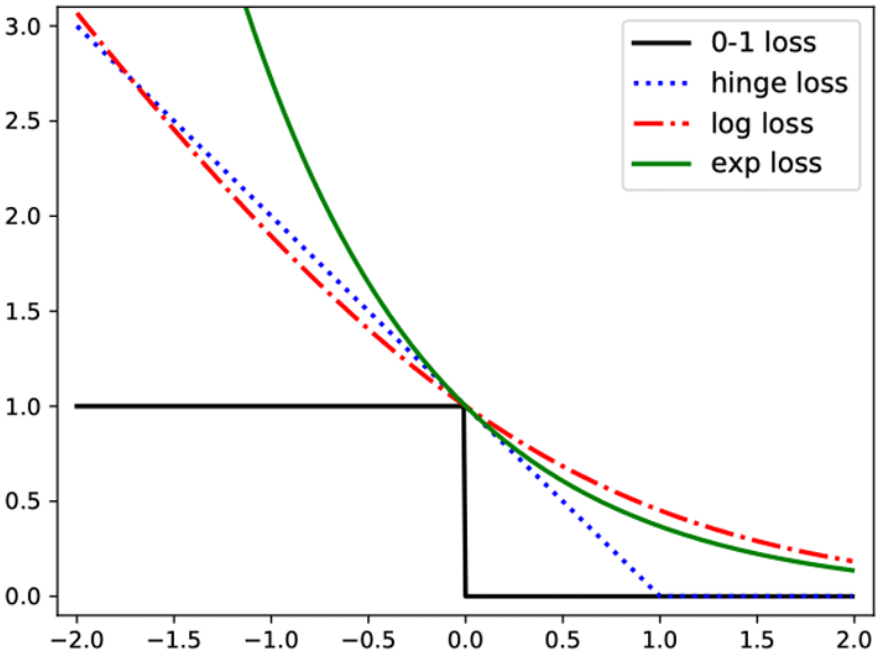



Figure 2: Surrogate loss functions compared to 0-1 loss. All surrogate losses upper bound the 0-1 loss while being differentiable and (mostly) convex.

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

1. **Upper bound** the 0-1 loss (so minimising the surrogate also reduces misclassification)
2. Be **convex** (guarantees finding global minimum)
3. Be **differentiable** (enables gradient-based optimisation)
4. Be "tight" (close to 0-1 loss, so the approximation is good)

### 21.3 Classification Errors

**Confusion Matrix**

| | | 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."

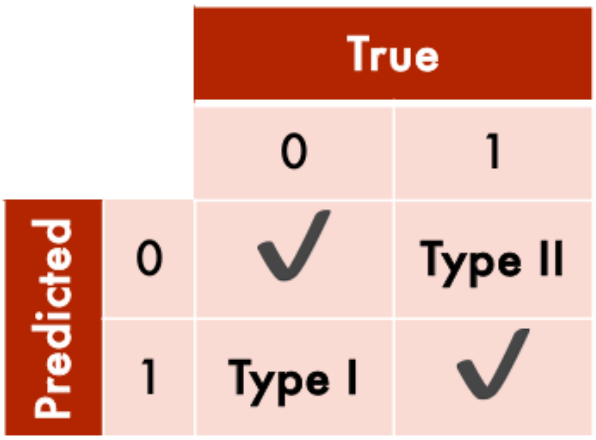



Figure 3: Confusion matrix structure showing the four possible outcomes of binary classification.

Different applications weight these errors differently:

- **Medical screening**: False negatives are dangerous (missing disease), so we tolerate more false positives
- **Spam filtering**: False positives are annoying (good email marked as spam), so we tolerate more false negatives
- **Criminal justice**: "Beyond reasonable doubt" means we prefer false negatives to false positives

## 22 Logistic Regression

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:

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

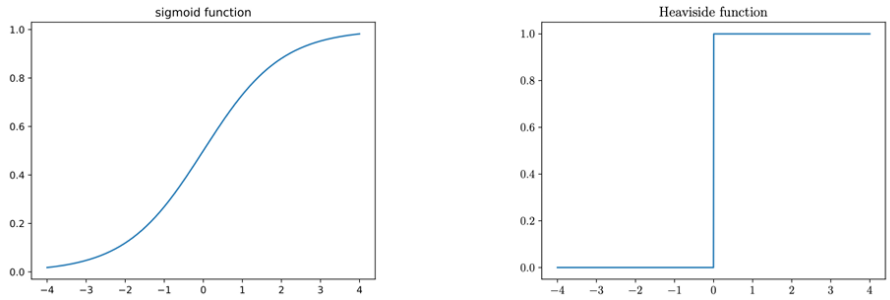



Figure 4: The sigmoid (logistic) function $\sigma(z) = 1/(1 + e^{-z})$ maps any real number to $(0, 1)$. At $z = 0$, $\sigma(0) = 0.5$. The function saturates at 0 and 1 for large $|z|$.

The sigmoid function has useful properties:

- Maps $\mathbb{R} \to (0, 1)$—valid probabilities
- Symmetric: $\sigma(-z) = 1 - \sigma(z)$
- Nice derivative: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$
- Monotonic: larger $x^\top \beta$ means higher probability

## 22.1 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:

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

where $\mu_i = \sigma(x_i^\top \beta)$.
Taking the negative log:

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

**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 $\mu$ depends nonlinearly on $\beta$ through the sigmoid.

**NB!**

[No Closed-Form Solution] Unlike linear regression, logistic regression has **no closed-form solution**. The dependence of $\mu_i$ on $\beta$ 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 $\beta$. We must use iterative optimisation:

- Gradient descent
- Newton's method (IRLS—Iteratively Reweighted Least Squares)
- Quasi-Newton methods (L-BFGS)

## 22.2 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.
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

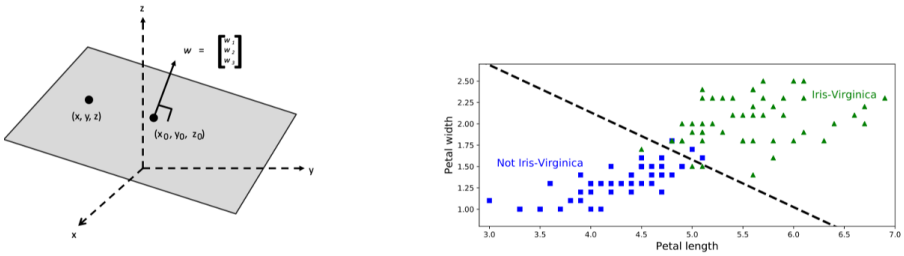



Figure 5: Linear decision boundary separating two classes. The boundary is the hyperplane where $x^\top \beta = 0$. Points on one side are classified as positive, points on the other as negative.

## 23 The Bias-Variance Tradeoff

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 an estimator $\hat{\theta}$ of true parameter $\theta$:
**Bias**: $\text{Bias}(\hat{\theta}) = \mathbb{E}[\hat{\theta}] - \theta$
How far is the average estimate from the truth?
**Variance**: $\text{Var}(\hat{\theta}) = \mathbb{E}[(\hat{\theta} - \mathbb{E}[\hat{\theta}])^2]$
How much does the estimate vary across different datasets?
**MSE Decomposition**:

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

**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 = Bias$^2$ + Variance)

**Key insight**: 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).

## 23.1 Example: Shrinkage Estimators

Consider estimating the mean $\mu$ 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: $\sigma^2/n$

**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 $\sigma/\sqrt{n}$ (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
- There is substantial uncertainty about the sample mean
- We wish to incorporate external information or beliefs into our estimation

**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

# 24 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**: $\hat{\beta} = (X^\top X)^{-1} X^\top y$; requires invertible $X^\top X$
6. **Variance of $\hat{\beta}$**: Sandwich form $(X^\top X)^{-1} X^\top \mathbb{E}[\epsilon\epsilon^\top] X (X^\top X)^{-1}$
7. **Robust SEs**: Handle heteroskedasticity via HC estimators; use squared residuals in the sandwich
8. **KL divergence**: MLE minimises KL divergence from true distribution to model
9. **Bayesian inference**: Parameters as random variables; posterior = likelihood $\times$ prior
10. **MAP**: MLE + prior = regularised estimation; Gaussian prior $\to$ Ridge, Laplace prior $\to$ Lasso
11. **Logistic regression**: Classification via sigmoid; trained with cross-entropy; no closed form
12. **Surrogate losses**: Differentiable, convex approximations to 0-1 loss
13. **Bias-variance**: MSE = Bias$^2$ + Variance; the tradeoff is fundamental to statistical learning

**ML Lecture Notes 2024**
Henry Baker

height

# ML Lecture Notes: Week 3
# High-Dimensional Methods & Regularisation

## 25 Supervised Learning: A Quick Recap

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

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 $(\hat{f}(x) - y)^2$.

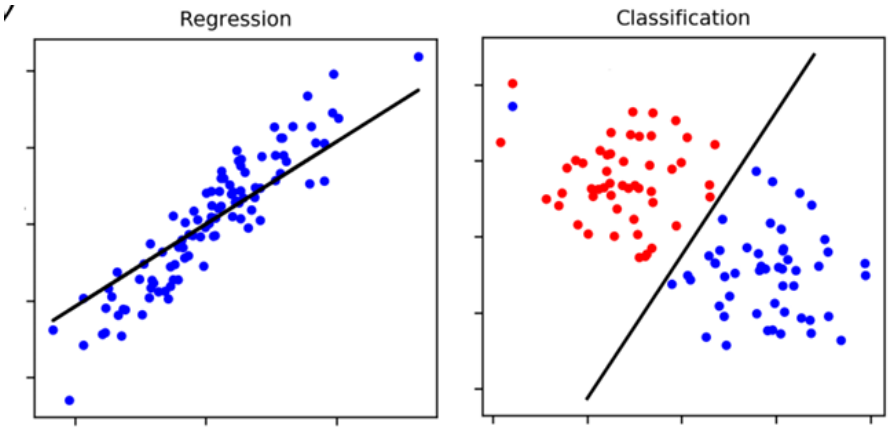



Figure 6: Regression vs classification in supervised learning. Left: regression fits a continuous function through the data. Right: classification finds a decision boundary separating classes.

### 25.1 OLS Recap

Recall that OLS gives us the optimal linear predictor:

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

This gives us the optimal coefficients; we then plug these 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**.

# 26 From Linear to Nonlinear: Polynomial Regression

**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** $\beta$—we are just using transformed features $[1, x, x^2, \ldots, x^M]$.
The design matrix becomes:

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

**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.

## 26.1 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)

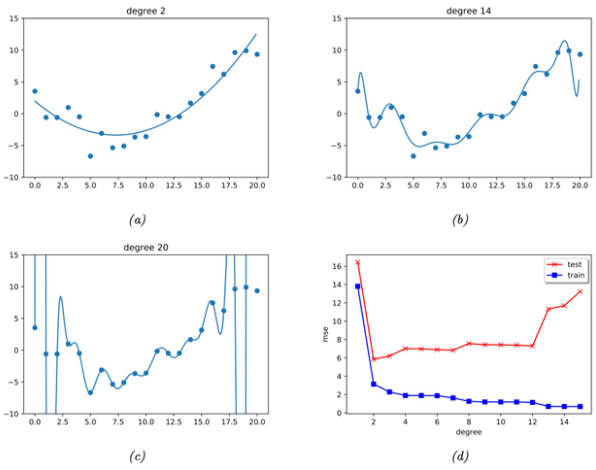



Figure 7: Effect of polynomial degree on fit. Higher degrees fit training data perfectly but generalise poorly. This illustrates the fundamental tension in model selection.

## 26.2 Why Polynomials Are Attractive (In Theory)

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

- **Taylor series**: Any smooth function can be approximated by its Taylor polynomial
- **Weierstrass approximation theorem**: Any continuous function on a closed interval can be uniformly approximated by polynomials to arbitrary precision

## 26.3 Challenges with Polynomial Regression: Numerical Instability

**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**

1. **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.

   *When $k$ gets large, $x^k$ becomes enormous.*

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

   *As $x^k$ explodes, $\beta_k$ 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.

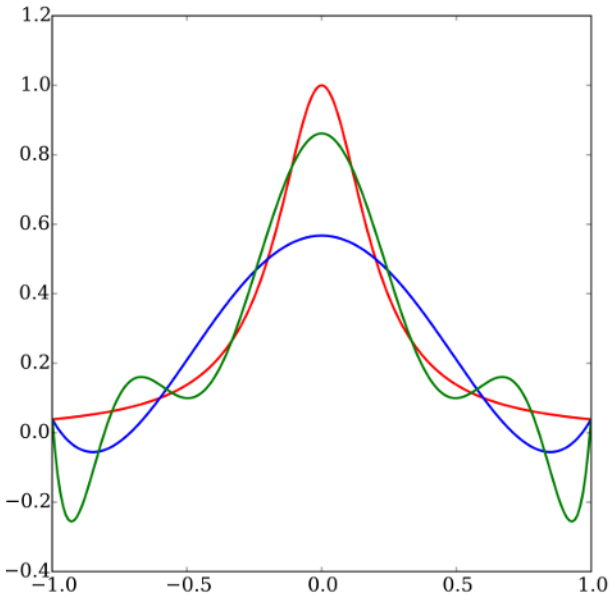



Figure 8: Numerical instability in high-degree polynomials: small perturbations in data cause large changes in the fitted curve, particularly near the edges of the domain.

### 26.3.1 The Condition Number

**Condition Number**

The **condition number** $\kappa(A) = \|A\| \cdot \|A^{-1}\|$ measures how sensitive $A^{-1}b$ is to perturbations in $b$.
For symmetric positive definite matrices:

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

- $\kappa \approx 1$: Well-conditioned, stable inversion
- $\kappa \gg 1$: Ill-conditioned, numerically unstable
- $\kappa = \infty$: Singular, non-invertible

The condition number of $X^\top X$ for polynomial regression grows rapidly with degree $M$. Intuitively, think of SVD: the condition number captures how "hard" it is to invert the matrix—it depends on the ratio between the largest and smallest singular values.

**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

# 27 Decomposing Prediction Error

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

## 27.1 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 $\theta$:

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

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

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

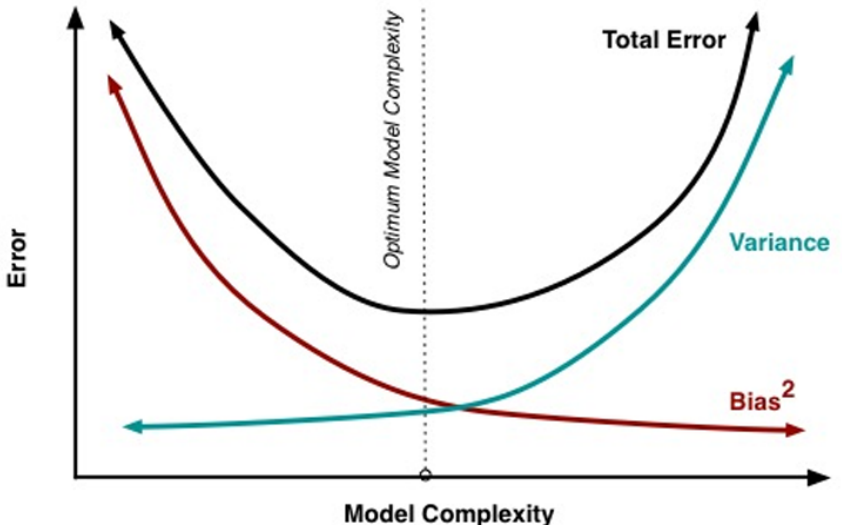



Figure 9: Bias-variance tradeoff: as model complexity increases, bias decreases but variance increases. The optimal complexity minimises total error (MSE). This U-shaped curve is fundamental to model selection.

## 27.2 Evaluation Metrics: Defining "Risk"

Different metrics capture different aspects of model performance. The choice of metric defines what "risk" we are 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**: $\frac{\text{TP}}{\text{TP+FP}}$. Use when the cost of false positives is high (e.g., spam filtering—you do not want legitimate emails marked as spam).
- **Recall**: $\frac{\text{TP}}{\text{TP+FN}}$. Use when the cost of false negatives is high (e.g., disease screening—you do not 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.

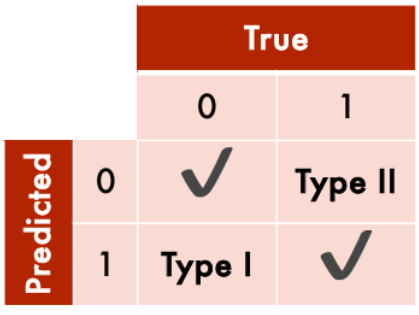



Figure 10: Confusion matrix showing Type I errors (false positives) and Type II errors (false negatives). These frame the loss in terms of incorrect predictions.

**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. For 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.

## 27.3 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 do not 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.

**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 is 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.

**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:**

- $\hat{f}_{\text{ERM}}$: 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...

## 27.4 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

## 27.5 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 $R_3 - R_1 = (R_2 - R_1) + (R_3 - R_2)$.

### 27.5.1 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.

### 27.5.2 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.

### 27.5.3 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.

## 27.6 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.

# 28 Regularisation

**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.

## 28.1 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)
- $\lambda \to \infty$: Ignore data, minimise complexity only

The parameter $\lambda$ manages the tradeoff between fitting the data and keeping the model simple.

## 28.2 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

## 28.3 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.

**Why Ridge Works**

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

### 28.3.1 Ridge as Rescaled OLS

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

$$\hat{\beta}_{\text{ridge}} = \frac{\hat{\beta}_{\text{OLS}}}{1 + \lambda}$$

Ridge uniformly shrinks all coefficients toward zero. This introduces bias but reduces variance.

## 28.4 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).

### 28.4.1 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 $\lambda$ from the absolute value
- $(\cdot)_+$: Sets result to zero if negative

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

**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 $X$ | $\frac{\hat{\beta}_{\text{OLS}}}{1+\lambda}$ | Soft thresholding |

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

### 28.5 Elastic Net

Combines L1 and L2 penalties:

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

where $\lambda_1$ and $\lambda_2$ control the impact of L1 and L2 terms respectively.
**Benefits**: Sparsity of Lasso + stability of Ridge. Particularly useful when features are correlated.

## 29 Multiple Perspectives on Regularisation

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

## 29.1 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

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 $\lambda I$ 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 $\lambda$ increases the diagonal terms, guaranteeing invertibility.

## 29.2 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 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).

## 29.3 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: $\lambda = \sigma^2/\tau^2$

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

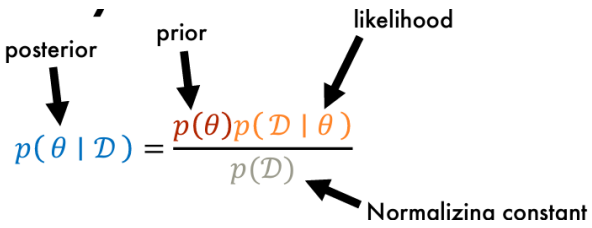



Figure 11: Bayes' theorem: the posterior $p(\theta|\mathcal{D})$ is proportional to the prior $p(\theta)$ times the likelihood $p(\mathcal{D}|\theta)$. Regularisation enters through the prior.

**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.

## 29.4 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** ($\ell_2$ ball)
- **Lasso**: Constraint region is a **diamond** ($\ell_1$ 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.

## 29.5 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 do not truly predict $y$ get shrunk because noisy versions of them show no relationship. Thus, adding noise is equivalent to regularisation.

# 30 Model Selection and Validation

How do we choose the regularisation strength $\lambda$? 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**.

## 30.1 Validation Sets

**Three-Way Split**

1. **Training set** ($p_{\text{train}}$): Fit model for each candidate $\lambda$
2. **Validation set** ($p_{\text{validation}}$): Choose $\lambda$ that minimises validation loss (model selection)
3. **Test set** ($p_{\text{test}}$): Estimate final generalisation error (used only once!)

This prevents "leaking" test information into model selection.

**NB!**

[Never Use the Test Set for Model Selection!] If you repeatedly evaluate on the test set and choose the best model, you are 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.

## 30.2 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 $k = 1, \ldots, K$:
   - 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.

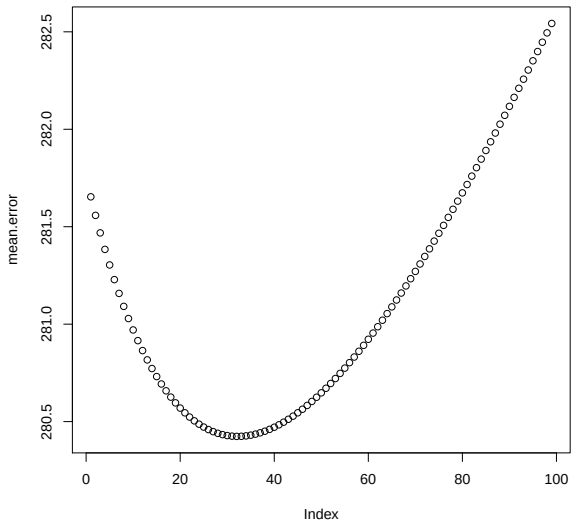



Figure 12: Cross-validation error as a function of model complexity (indexed on x-axis). The U-shaped curve shows the bias-variance tradeoff: too simple models underfit (high error on left), too complex models overfit (high error on right). The optimal complexity minimises CV error.

**Choosing $\lambda$ via CV**

1. Define a grid of $\lambda$ values (e.g., $10^{-4}, 10^{-3}, \ldots, 10^{2}$)
2. For each $\lambda$, compute CV score
3. Select $\lambda^* = \arg\min_\lambda \text{CV}(\lambda)$
4. Refit on full training data with $\lambda^*$

## 31 Regularised Polynomial Regression

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 $\lambda$ via cross-validation

This allows fitting smooth, complex curves without the instability of high-degree unregularised polynomials.

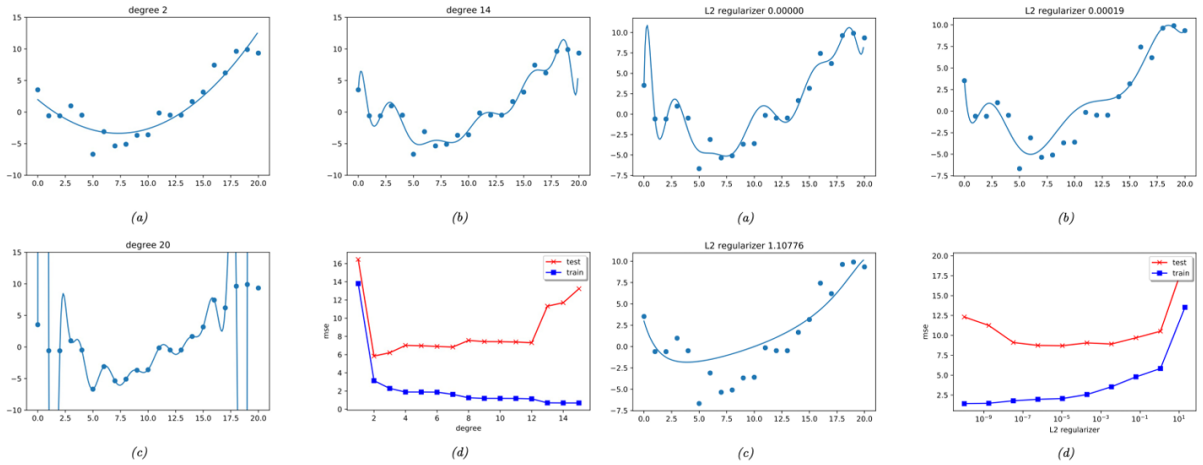



Figure 13: Regularised polynomial regression. The right panels show L2 regularisation producing smooth fits even with high-degree polynomials, avoiding the oscillatory behaviour of unregularised fits. Panel (c) is particularly notable: a smooth model that does not exhibit typical polynomial wiggles. Regularisation allows us to get the best of both worlds—expressivity without instability.

## 32 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 $X^\top X$; global approximators cause edge problems (Runge's phenomenon)
3. **Population vs empirical risk**: We minimise the latter (what we can compute) to approximate the former (what we care about)
4. **Three levels of optimality**: Bayes optimal $f^{**}$, best-in-class $f^*$, empirical best $\hat{f}_n$
5. **Approximation error**: Limitation of hypothesis class (does not decrease with $n$)
6. **Estimation error**: Finite-sample error (decreases with $n$, increases with complexity)
7. **Regularisation**: Penalty on complexity to reduce overfitting; trades bias for variance
8. **Ridge (L2)**: Shrinks coefficients, guarantees invertibility, closed-form solution
9. **Lasso (L1)**: Sparse solutions, automatic feature selection, no closed-form
10. **Multiple perspectives**: Regularisation as necessity, bias-variance tradeoff, Bayesian prior, geometric constraint, measurement error
11. **Bayesian view**: Ridge = Gaussian prior, Lasso = Laplace prior
12. **Cross-validation**: Choose hyperparameters without overfitting to test data

## Overview

This week addresses the fundamental question: *how well will our model perform on data it has never seen?* We develop both practical tools (cross-validation) and theoretical frameworks (generalisation bounds, VC dimension) for reasoning about this question.
**Key themes:**

- Practical validation strategies for hyperparameter selection
- Theoretical bounds on generalisation error
- Measuring hypothesis class complexity: VC dimension
- Generalisation behaviour in low and high dimensional regimes

# 33 Cross-Validation

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.

### 33.1 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—that would leak information and invalidate our final evaluation. Cross-validation solves this by cleverly reusing the training data.

### 33.2 K-Fold Cross-Validation

**K-Fold Cross-Validation**

Partition the training data into $K$ roughly equal-sized **folds**. For each fold $k \in \{1, \ldots, K\}$:

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:

$$R^{\text{cv}} = \frac{1}{K} \sum_{k=1}^{K} R\left(\hat{\theta}(D_{-k}), D_k\right) \tag{1}$$

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

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.

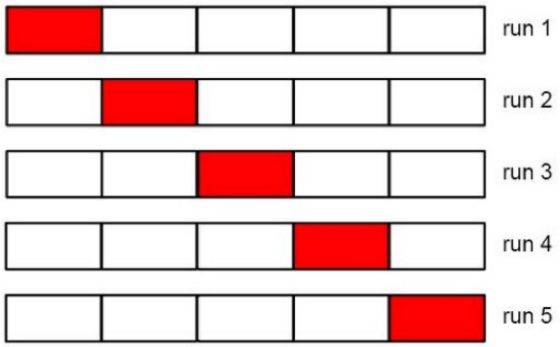



Figure 14: 5-fold cross-validation: the data is split into 5 folds, and each fold serves as the validation set exactly once while the remaining 4 folds form the training set.

**Common Choices for $K$**

- $K = 5$ **or** $K = 10$: Standard choices that balance bias and variance of the CV estimate. $K = 10$ is perhaps most common in practice.
- $K = n$ **(Leave-One-Out CV, LOOCV)**: Each fold contains exactly one observation. Maximises use of training data but is computationally expensive ($n$ model fits) and can have high variance.

**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.

## 33.3 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 "amazing" properties of linear regression.

**LOOCV Shortcut for Linear Regression**

Define the **hat matrix** (or projection matrix):

$$H = X(X^\top X)^{-1}X^\top \tag{2}$$

The hat matrix projects the observed responses onto the fitted values: $\hat{y} = Hy$.
The LOOCV mean squared error can be computed directly as:

$$\text{MSE}_{\text{cv}} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{y}_i}{1 - h_{ii}}\right)^2 = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{e_i}{1 - h_{ii}}\right)^2 \tag{3}$$

where:

- $y_i$ is the actual response for observation $i$
- $\hat{y}_i$ is the predicted value from the full model (fitted on all $n$ points)
- $e_i = y_i - \hat{y}_i$ is the ordinary residual
- $h_{ii}$ is the $i$th diagonal element of $H$, called the **leverage** of observation $i$

**Why does this work?** The leverage $h_{ii}$ measures how much observation $i$ influences its own prediction. When we leave out observation $i$ and refit, the prediction at that point changes. The factor $(1 - h_{ii})$ in the denominator exactly corrects for this change, transforming the ordinary residual into what the residual *would have been* if we had actually left out observation $i$.
**Interpretation of leverage**: High-leverage points are observations where $x_i$ is far from the centre of the predictor space. These points have more "pull" on the regression line. The leverage satisfies $0 \leq h_{ii} \leq 1$, with $\sum_i h_{ii} = p$ (the number of predictors including the intercept).

**Significance of the LOOCV Shortcut**

- Allows direct calculation of LOOCV error without refitting $n$ separate models
- Leverages the mathematical structure of linear regression (specifically the hat matrix)
- Makes LOOCV computationally attractive for linear models
- Same computational cost as fitting the model once

## 33.4 The 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 $\hat{R}_\lambda$ for each hyperparameter setting $\lambda$
2. **Find the minimum**: Let $\hat{R}_{\min}$ be the lowest CV error observed
3. **Compute the standard error**:

$$\text{SE} = \frac{\sigma_{\text{cv}}}{\sqrt{K}} \tag{4}$$

   where $\sigma_{\text{cv}}$ is the standard deviation of the $K$ fold-wise error estimates

4. **Select the simplest model** whose CV error satisfies:

$$\hat{R} \leq \hat{R}_{\min} + \text{SE} \tag{5}$$

**Why prefer simpler models?** This rule implements **Occam's razor**: when multiple models have statistically indistinguishable performance (within one standard error), we prefer the simpler one. Simpler models are:

- More interpretable
- Less prone to overfitting
- More likely to generalise to genuinely new situations

By acknowledging uncertainty in our risk estimates, we recognise that a slightly more regularised (simpler) model is within reasonable margin of error as good as the very lowest—and since we have uncertainty, we should favour the simpler one.

## 33.5 The Optimism of Training Error

**NB!**

[Training Error is Optimistically Biased] Training error systematically **underestimates** true out-of-sample error. The model parameters $\theta$ 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:

$$\hat{\lambda} = \arg\min_\lambda \min_\theta R_\lambda(\theta, D) \tag{6}$$

$$= \arg\min_\lambda \left[ \min_\theta R(\theta, D) + \lambda C(\theta) \right] \tag{7}$$

$$= 0 \tag{8}$$

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.

This phenomenon—where performance on training data is better than on test data—is called the **optimism** of the training error. Cross-validation corrects for this optimism by ensuring that model evaluation always occurs on held-out data.

### 33.6 Grouped Cross-Validation

Standard K-fold CV assumes observations are independent and identically distributed (i.i.d.). When this assumption is violated, naive CV can cause **information leakage**, leading to overoptimistic estimates.

**Violations of Independence**

Consider a linear model $y_i = X_i\beta + \epsilon_i$ where the errors are correlated within groups:

$$\epsilon \sim \mathcal{N}(0, \Sigma) \quad \text{with} \quad \Sigma_{ij} \neq 0 \text{ if } c(i) = c(j) \tag{9}$$

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.

- **Examples:**

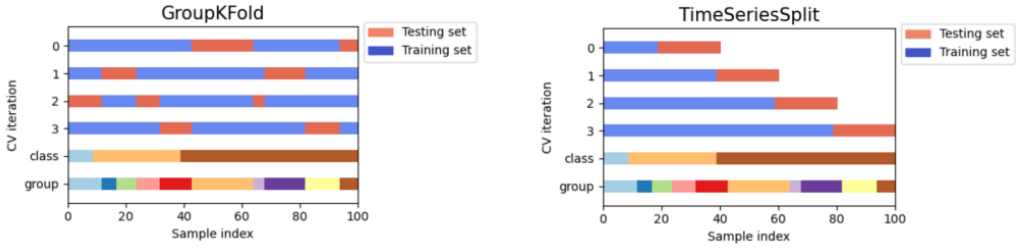



Figure 15: **Left**: Group K-fold keeps all observations from a given group together in the same fold across all CV iterations. **Right**: Time series split respects temporal ordering, always using past data to predict future data.

**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)
- You want to predict for *new groups* not seen during training

**Time Series Split**: Training set grows over time; test set is always in the future. Use when:

- Data has temporal structure
- You want to predict future observations from past data
- **Never** use future data to predict the past

**General Principle**: The CV split should mirror how the model will be deployed. If you will predict for new groups, use group CV. If you will 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.

## 34 Frequentist vs Bayesian Risk

The definitions of risk we have used so far are **frequentist**. It is worth contrasting this with the Bayesian perspective.

**Frequentist Risk**

$$R(\theta, f) = \mathbb{E}_{p(x|\theta)}[\ell(\theta, f(x))] \tag{10}$$

**Key features**:

- Parameters $\theta$ are **fixed but unknown** constants
- The expectation is over the **data** $x$ drawn from $p(x|\theta)$
- Risk measures average loss across repeated sampling from the same data-generating process
- Data is the random variable; parameters are fixed

**Bayes Risk**

$$R_{\pi_0}(f) = \mathbb{E}_{\pi_0(\theta)}[R(\theta, f)] = \int \pi_0(\theta)\, p(x|\theta)\, \ell(\theta, f(x))\, d\theta\, dx \tag{11}$$

**Key features**:

- Parameters $\theta$ are treated as a **random variable** with prior distribution $\pi_0(\theta)$
- The expectation is over **both** $\theta$ 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 $\theta$, determine the likelihood of observing data $x$
2. Compute the loss for decision function $f$ given that $\theta$ and $x$
3. Weight this loss by the joint probability of $\theta$ and $x$ (coming from prior $\times$ likelihood)

4. Integrate (sum) across all possible $\theta$ 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 |

# 35 Generalisation Bounds

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.

## 35.1 Error Decomposition Recap

**Error Decomposition**

Recall the fundamental decomposition of excess risk:

$$\underbrace{\mathbb{E}[\mathcal{R}(f_n^*)] - \mathcal{R}(f^{**})}_{\text{Total excess risk}} = \underbrace{\mathcal{R}(f^*) - \mathcal{R}(f^{**})}_{\text{Approximation error}} + \underbrace{\mathbb{E}[\mathcal{R}(f_n^*)] - \mathcal{R}(f^*)}_{\text{Estimation error}} \tag{12}$$

where:

- $f^{**} = \arg\min_f \mathcal{R}(f)$: The **Bayes optimal** predictor—best possible function
- $f^* = \arg\min_{f \in \mathcal{H}} \mathcal{R}(f)$: 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

**Approximation error** measures the cost of restricting to hypothesis class $\mathcal{H}$—how much we lose by not considering all possible functions. **Estimation error** measures the cost of learning from finite data—how far our learned $f_n^*$ is from the best-in-class $f^*$.

**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 $\beta_0^*, \beta_1^*$ 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 $\mathcal{H}$?

## 35.2 Uses of Bounds

**NB!**

[Bounds Are Not Replacements for Empirical Validation] Generalisation bounds are typically too loose to be directly useful for predicting 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.

## 35.3 Building Blocks: Concentration Inequalities

To derive generalisation bounds, we need tools that quantify how sample averages concentrate around their expectations.

**Hoeffding's Inequality**

For $X_1, \ldots, X_n \overset{\text{iid}}{\sim} \text{Bernoulli}(\mu)$, the sample mean $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$ satisfies:

$$P\left(|\bar{X} - \mu| > \epsilon\right) \leq 2\exp(-2n\epsilon^2) \tag{13}$$

**Interpretation**: The probability that the sample mean deviates from the true mean by more than $\epsilon$ decreases **exponentially** in $n$ (sample size) and $\epsilon^2$ (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

**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.

**Union Bound (Boole's Inequality)**

For any events $\mathcal{E}_1, \ldots, \mathcal{E}_d$:

$$P\left(\bigcup_{i=1}^{d} \mathcal{E}_i\right) \leq \sum_{i=1}^{d} P(\mathcal{E}_i) \tag{14}$$

The probability that *at least one* event occurs is at most the sum of the individual probabilities.

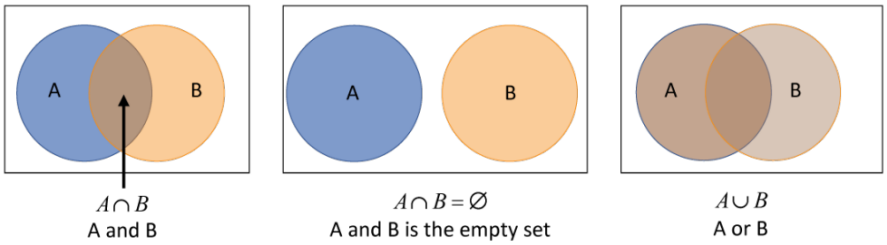



Figure 16: Union bound: $P(A \cup B) \leq P(A) + P(B)$. The bound is loose when events overlap significantly (we double-count the intersection), but it avoids computing complex intersections.

The union bound is useful because computing exact probabilities of unions is hard when events have complex dependencies. It provides a conservative (loose) but easy-to-compute upper bound.

## 35.4 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 $\mathcal{H}$:

$$P\left(\max_{f \in \mathcal{H}} |\mathcal{R}(f) - \hat{\mathcal{R}}_n(f)| > \epsilon\right) \leq 2|\mathcal{H}| \cdot \exp(-2n\epsilon^2) \tag{15}$$

Equivalently: with probability at least $1 - \delta$, for all $f \in \mathcal{H}$ simultaneously:

$$|\mathcal{R}(f) - \hat{\mathcal{R}}_n(f)| \leq \sqrt{\frac{\log(2|\mathcal{H}|/\delta)}{2n}} \tag{16}$$

**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 $\mathcal{H}$.

**Proof Sketch**

$$P\left(\max_{f\in\mathcal{H}}|\mathcal{R}(f)-\hat{\mathcal{R}}_n(f)|>\epsilon\right) = P\left(\bigcup_{f\in\mathcal{H}}\{|\mathcal{R}(f)-\hat{\mathcal{R}}_n(f)|>\epsilon\}\right) \tag{17}$$

$$\leq \sum_{f\in\mathcal{H}} P\left(|\mathcal{R}(f)-\hat{\mathcal{R}}_n(f)|>\epsilon\right) \quad \text{(Union bound)} \tag{18}$$

$$\leq \sum_{f\in\mathcal{H}} 2\exp(-2n\epsilon^2) \quad \text{(Hoeffding for each } f\text{)} \tag{19}$$

$$= 2|\mathcal{H}|\cdot\exp(-2n\epsilon^2) \quad \text{(Finiteness of } \mathcal{H}\text{)} \tag{20}$$

**Interpreting the Bound**

- $|\mathcal{H}|$ **(hypothesis space size)**: Larger hypothesis spaces give looser bounds. More models ⇒ more chances to overfit ⇒ need more data.
- $\exp(-2n\epsilon^2)$ **(sample size effect)**: More data ⇒ tighter bounds. Improvement is exponential in $n$.
- $\epsilon$ **(tolerance)**: Smaller tolerance ⇒ looser bounds. It is harder to guarantee small errors.

**The fundamental tradeoff**: Error in the population increases with hypothesis space size but decreases with sample size.

### 35.5 Limitations of This Bound

1. **Finite hypothesis class required**: What about continuous parameters? Linear regression has infinitely many possible $\beta$ 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 may be much better than the bound suggests.

## 36 Measuring Hypothesis Class Complexity

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?

**Approaches to Measuring Complexity**

- **Parameter counting**: Number of free parameters (degrees of freedom)
- **Smoothness measures**: Derivative-based measures (e.g., Sobolev norms), quantifying "wiggliness"
- **VC dimension**: Largest set of points that can be "shattered"
- **Rademacher complexity**: Ability to fit random noise

## 36.1 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 $\theta$)
- 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 $\sim 50$ 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 (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 $\beta_i$ 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.

## 36.2 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 $\{x_1, \ldots, x_n\}$ if, for every possible labelling $(y_1, \ldots, y_n) \in \{0,1\}^n$, there exists some $f \in \mathcal{H}$ that correctly classifies all points:

$$f(x_i) = y_i \quad \text{for all } i = 1, \ldots, n \tag{21}$$

In other words, $\mathcal{H}$ can achieve zero training error on these $n$ points regardless of how they are labelled.

**VC Dimension**

The **VC dimension** $\mathrm{VC}(\mathcal{H})$ is the largest number of points that can be shattered by $\mathcal{H}$:

$$\mathrm{VC}(\mathcal{H}) = \max\{n : \exists \{x_1, \ldots, x_n\} \text{ that } \mathcal{H} \text{ can shatter}\} \tag{22}$$

If $\mathcal{H}$ can shatter arbitrarily large sets, we say $\mathrm{VC}(\mathcal{H}) = \infty$.

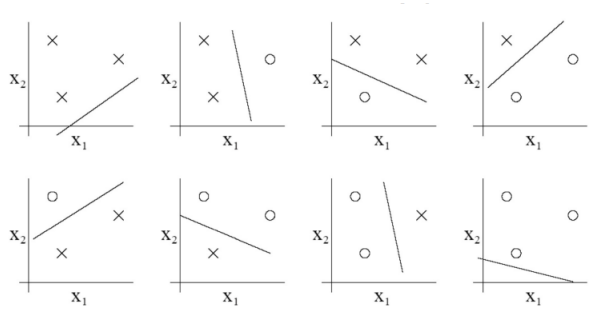



Figure 17: Linear classifiers in 2D can shatter 3 points in general position: for any labelling of the 3 points, we can find a line that separates them correctly. Hence VC(linear classifiers in $\mathbb{R}^2$) $\geq 3$.

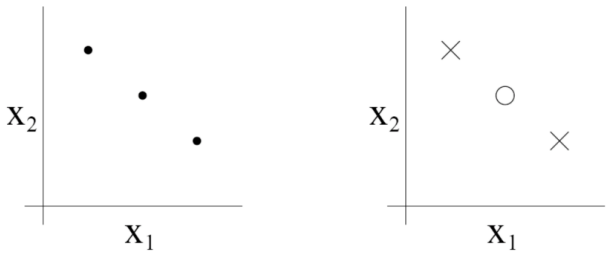



Figure 18: However, 4 points in general position *cannot* be shattered by linear classifiers in 2D. The "XOR" labelling (opposite corners have the same label) cannot be achieved by any line. Hence VC(linear classifiers in $\mathbb{R}^2$) $= 3$.

**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!

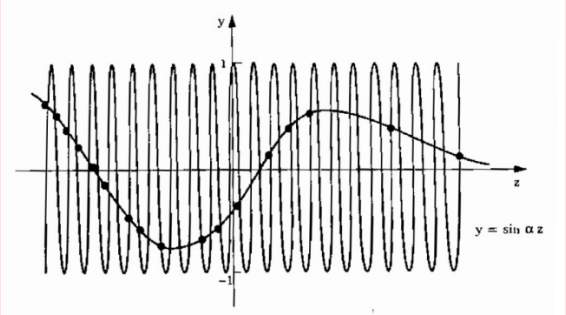



Figure 19: The function $f(x) = \text{sign}(\sin(\omega x))$ has only **one parameter** ($\omega$) but **infinite VC dimension**. By choosing $\omega$ large enough, the function can oscillate rapidly enough to shatter arbitrarily many points on the real line.

**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

## 36.3 The VC Bound

The VC dimension allows us to state generalisation bounds for infinite hypothesis classes.

**VC Generalisation Bound**

With probability at least $1 - \delta$:

$$\mathcal{R}(f_n^*) - \mathcal{R}(f^*) \leq \sqrt{\frac{1}{n}\left[V\left(\log\frac{2n}{V} + 1\right) - \log\frac{\delta}{4}\right]} \tag{23}$$

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.

# 37 Structural Risk Minimisation

Given generalisation bounds, one might consider directly minimising them rather than using cross-validation:

$$\hat{f} = \arg\min_{f \in \mathcal{H}} \left[ \hat{\mathcal{R}}_n(f) + \text{complexity penalty} \right] \tag{24}$$

This is **structural risk minimisation** (SRM). The idea is to balance empirical performance against hypothesis class complexity in a principled way derived from theory.
**In practice**: Cross-validation usually works better because:

- Theoretical bounds are often very loose
- Cross-validation adapts to the actual data distribution
- SRM requires knowing the VC dimension, which can be hard to compute

However, SRM provides theoretical justification for regularisation: adding a complexity penalty to the loss function is exactly what the theory recommends.

# 38 Generalisation in Linear Regression

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.

## 38.1 OLS Estimation Error

**OLS Estimator Decomposition**

The OLS estimator $\hat{\beta} = (X^\top X)^{-1} X^\top y$ can be decomposed as:

$$\hat{\beta} = \beta^* + (X^\top X)^{-1} X^\top \epsilon \tag{25}$$

where $\beta^*$ are the true parameters and $\epsilon$ is the noise vector.
The **estimation error** $\hat{\beta} - \beta^*$ thus equals $(X^\top X)^{-1} X^\top \epsilon$—it depends on:

- The design matrix $X$ (specifically, how well-conditioned $X^\top X$ is)
- The noise $\epsilon$

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.

## 38.2 Singular Value Decomposition (SVD)

To analyse generalisation in different dimensional regimes, we need the singular value decomposition.

**Singular Value Decomposition**

Any $n \times p$ matrix $X$ of rank $r$ can be decomposed as:

$$X = U\Sigma V^\top \tag{26}$$

where:

- $U$ is $n \times r$ with orthonormal columns: $U^\top U = I_r$
- $\Sigma$ is $r \times r$ diagonal with singular values $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$
- $V$ is $p \times r$ with orthonormal columns: $V^\top V = I_r$

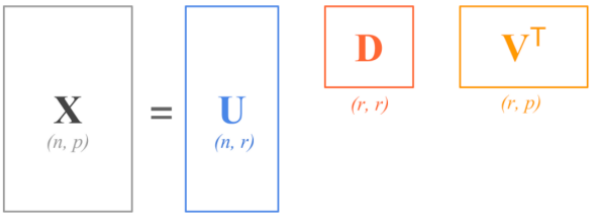



Figure 20: SVD dimensions: $X$ is $n \times p$, decomposed into $U$ $(n \times r)$, $\Sigma$ $(r \times r)$, and $V^\top$ $(r \times p)$.

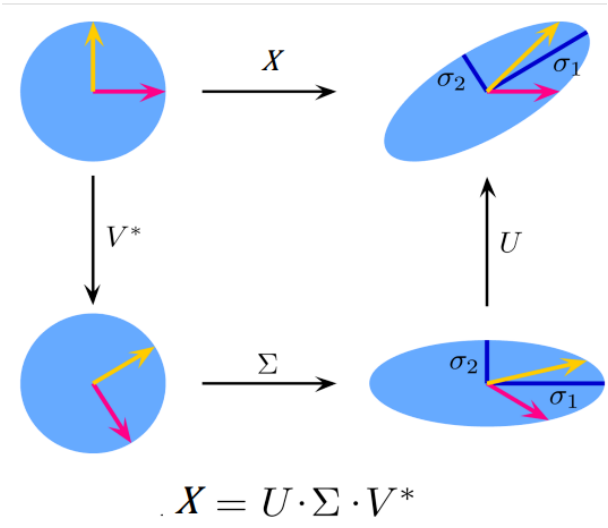



Figure 21: Geometric interpretation of SVD: $V^\top$ rotates input vectors, $\Sigma$ scales along principal axes, $U$ rotates to output space. SVD decomposes any linear transformation into rotation-scale-rotation.

**Key SVD Properties**

- $X^\top X = V\Sigma^2 V^\top$ (eigendecomposition of $X^\top X$)
- $XX^\top = U\Sigma^2 U^\top$ (eigendecomposition of $XX^\top$)
- OLS predictions on training data: $\hat{y} = X\hat{\beta} = UU^\top y$
- Predictions on new data $\tilde{X}$: $\tilde{X}\hat{\beta} = \tilde{X}V\Sigma^{-1}U^\top y$

SVD provides a numerically stable way to compute the pseudo-inverse, even when $X^\top X$ 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 $\sigma_i$). Small singular values indicate near-collinearity, which inflates estimation variance.

# 39 Low vs High Dimensional Regimes

The behaviour of OLS depends critically on the relationship between $p$ (number of features) and $n$ (number of observations).

## 39.1 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:

$$R(\hat{\beta}) - R(\beta^*) = \frac{p}{n}\sigma^2 \tag{27}$$

where $\sigma^2$ is the noise variance.

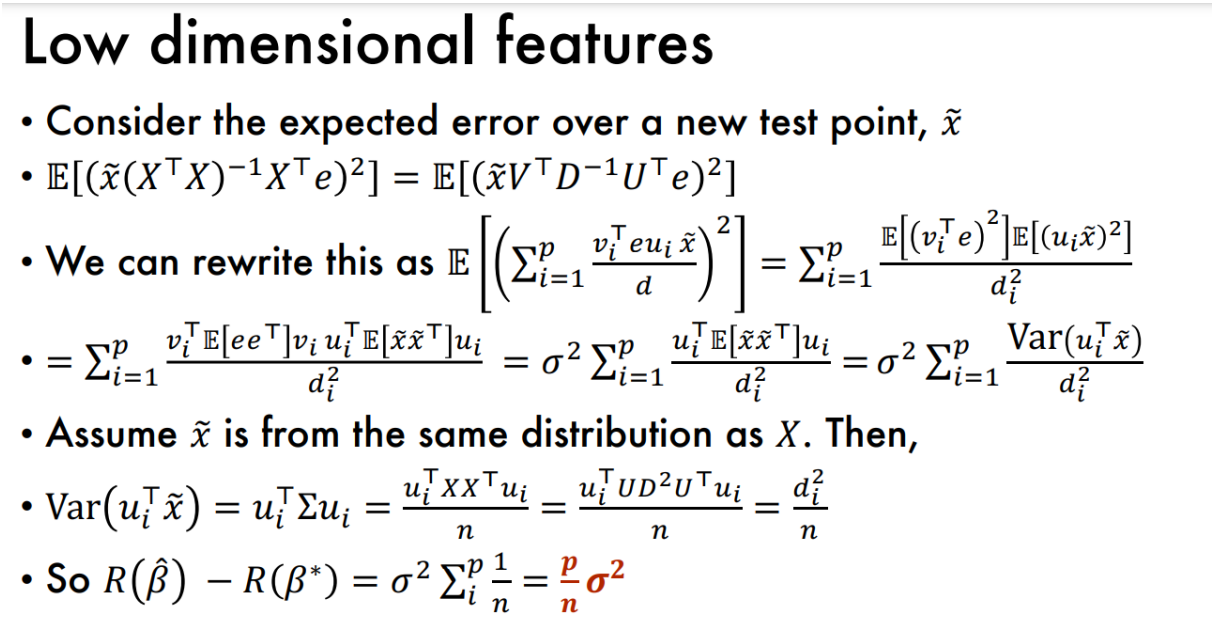



Figure 22: Low-dimensional regime: as $n$ increases, the excess risk decreases smoothly. More data leads to rapid improvement in generalisation.

**Interpretation**:

- Risk scales linearly with $p$: more parameters ⇒ more estimation error
- Risk scales inversely with $n$: more data ⇒ less error
- The ratio $p/n$ is the key quantity
- Adding data provides **rapid** improvement

### 39.2 High-Dimensional Regime: $p > n$

When $p > n$, the system is underdetermined—there are infinitely many $\beta$ that perfectly interpolate the training data. OLS as typically defined fails ($(X^\top X)^{-1}$ does not exist), but the **minimum-norm** (Moore-Penrose pseudo-inverse) solution can still be computed.

**Expected Risk in High Dimensions**

When $p > n$ using the minimum-norm OLS solution:

$$R(\hat{\beta}) - R(\beta^*) \approx \left(1 - \frac{n}{p}\right)\|\beta^*\|^2 + \frac{n}{p}\sigma^2 \tag{28}$$

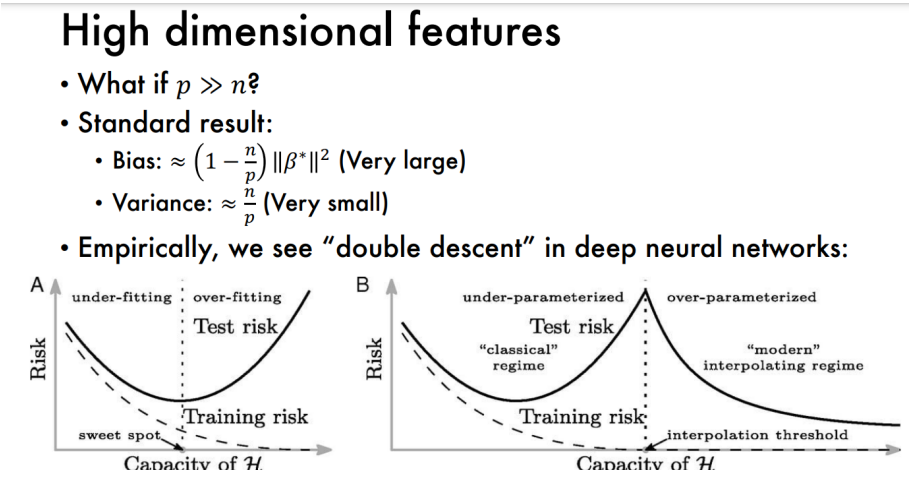



Figure 23: High-dimensional regime: the relationship between model complexity and error follows different dynamics. Note the different scaling compared to the low-dimensional case.

**Interpretation**:

- The first term $(1 - n/p)\|\beta^*\|^2$ dominates when $p \gg n$—this is essentially **bias**
- Adding more data ($n$) helps only marginally when $p$ is very large
- The minimum-norm solution has low variance but high bias

**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 | $p/n$ | $n/p$ |

## 39.3 The Interpolation Threshold and Double Descent

Something interesting happens around $p \approx n$—the **interpolation threshold**. Classical theory predicts that test error should peak here, and indeed it does. But what happens beyond?

**Double Descent Phenomenon**

Classical U-shaped bias-variance curves predict that test error increases 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)

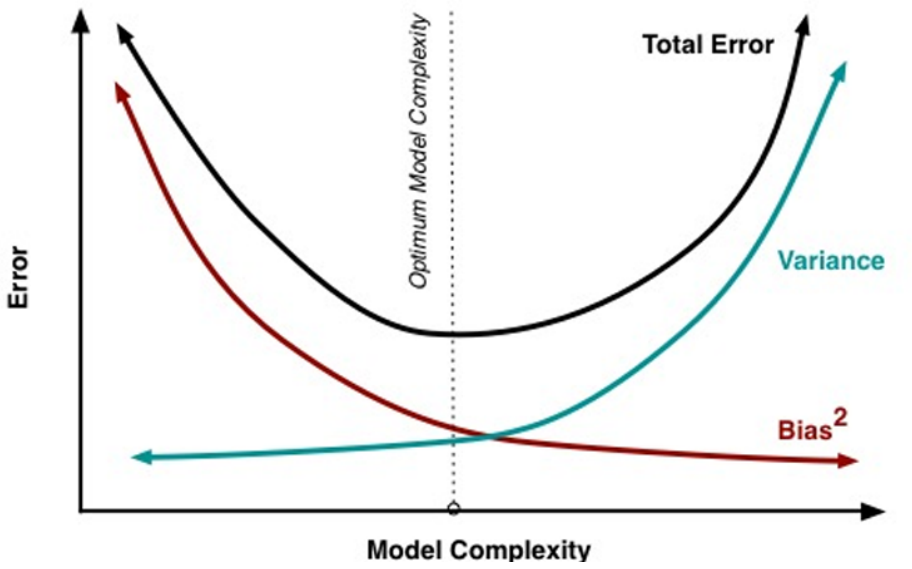



Figure 24: Classical bias-variance tradeoff: total error (black) is the sum of squared bias (red, decreasing with complexity) and variance (blue, increasing with complexity). The optimal complexity balances these. Double descent challenges this picture in the overparameterised regime.

**Why does this happen?** In the overparameterised regime:

- The model can perfectly fit (interpolate) the training data
- Among all interpolating solutions, gradient descent tends to find the **minimum-norm** solution
- This implicit regularisation keeps the solution "simple" despite the high parameter count
- As $p$ increases further, the minimum-norm solution becomes smoother and generalises better

This is an active area of research. The key insight is that **parameter count alone does not determine generalisation**—the specific solution found by the learning algorithm matters enormously.

**NB!**

[Implications for Deep Learning] Double descent helps explain why deep neural networks with millions of parameters can generalise well despite classical theory predicting overfitting:

- They operate 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.

## 40 Bias-Variance Decomposition

The expected prediction error can be decomposed into interpretable components.

**Bias-Variance-Noise Decomposition**

For squared error loss, the expected prediction error at a new point $x$ can be written:

$$\mathbb{E}[(y - \hat{f}(x))^2] = \underbrace{\text{Bias}[\hat{f}(x)]^2}_{\text{Systematic error}} + \underbrace{\text{Var}[\hat{f}(x)]}_{\text{Estimation variability}} + \underbrace{\sigma^2}_{\text{Irreducible noise}} \tag{29}$$

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
- $\sigma^2$ is the inherent noise in the data-generating process

**The tradeoff**:

- **Simple models** (few parameters, strong regularisation): High bias, low variance
- **Complex models** (many parameters, weak regularisation): Low bias, high variance

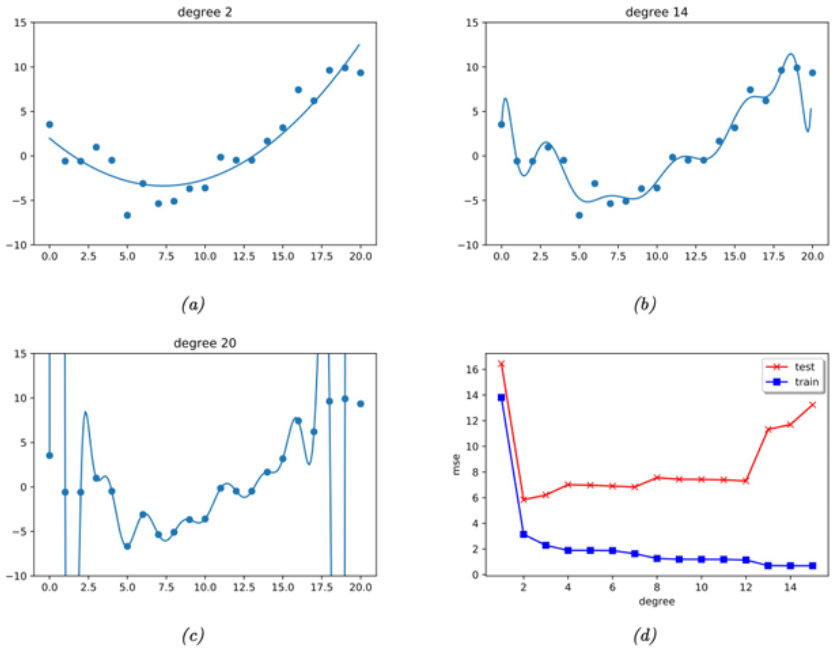



Figure 25: Effect of model complexity on fit. (a) Low complexity (degree 2): underfitting, high bias. (b) Medium complexity (degree 14): good fit. (c) High complexity (degree 20): overfitting, high variance. (d) Training error decreases monotonically with complexity; test error is U-shaped.

**Bias-Variance in Different Regimes**

**Underparameterised** ($p < n$):

- Increasing complexity: bias $\downarrow$, variance $\uparrow$
- Optimal complexity balances the two

**At interpolation threshold** ($p \approx n$):

- Variance can become very large
- Small changes in data cause large changes in the fit

**Overparameterised** ($p \gg n$):

- With implicit regularisation: both bias and variance can be low
- This is the "benign overfitting" regime

# 41 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
   - LOOCV has a closed-form shortcut for linear regression
   - One standard error rule implements Occam's razor
   - Use grouped/time-series variants for non-i.i.d. data
2. **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 loose but provide conceptual insight
3. **VC dimension**: Measures hypothesis class complexity via shattering
   - Does not always equal parameter count
   - Enables bounds for infinite hypothesis classes
4. **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
   - High-dim: bias dominates, more data helps marginally
   - Double descent challenges classical bias-variance intuition
5. **Bias-variance tradeoff**: Fundamental decomposition of prediction error
   - Simple models: high bias, low variance
   - Complex models: low bias, high variance
   - Overparameterised models: can achieve low bias *and* low variance with implicit regularisation

## Overview

This week addresses a surprising phenomenon: models that perfectly fit training data (including noise) can still generalise well. We develop the theoretical framework for understanding when and why "benign overfitting" occurs, connecting the practical success of overparameterised models like deep neural networks to rigorous analysis using singular value decomposition.
**Key themes:**

- Contrasting behaviour of OLS in low vs high dimensional regimes
- The double descent phenomenon: why test error can *decrease* beyond the interpolation threshold
- The manifold hypothesis: why high-dimensional data often has low intrinsic structure
- SVD-based analysis of benign overfitting
- Conditions under which interpolating models generalise well

# 42 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).

**Relationship Between Risk**

Recall the key relationships:

- **Risk** = expected loss over the data distribution
- 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

## 42.1 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)
**Risk formula**:

$$R(\hat{\beta}) - R(\beta^*) = \frac{p}{n}\sigma^2 \tag{30}$$

**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: $\mathbb{E}[\hat{\beta}] = \beta^*$

**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.

## 42.2 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**:

$$R(\hat{\beta}) - R(\beta^*) \approx \left(1 - \frac{n}{p}\right)\|\beta^*\|^2 + \frac{n}{p}\sigma^2 \tag{31}$$

This decomposes into:

- **Bias**: $\approx (1 - \frac{n}{p})\|\beta^*\|^2$ — very large when $p \gg n$
- **Variance**: $\approx \frac{n}{p}\sigma^2$ — very small when $p \gg n$

**Understanding the high-dimensional decomposition**:
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 too many degrees of freedom relative to the data
- It can fit noise perfectly, leading to systematic errors on new data
- The bias depends on $\|\beta^*\|^2$—larger true coefficients mean larger potential bias

The variance term $\frac{n}{p}\sigma^2$ is small because:

- The model is highly constrained by having so many parameters "explain" relatively few observations

- There is little room for the estimates to vary across different samples
- Counterintuitively, having more parameters leads to *lower* variance (but much higher bias)

**On the Margin**

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
- 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

## 42.3 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 |
| OLS status | BLUE (optimal) | Problematic without regularisation |

## 42.4 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.

# 43 The Double Descent Phenomenon

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.

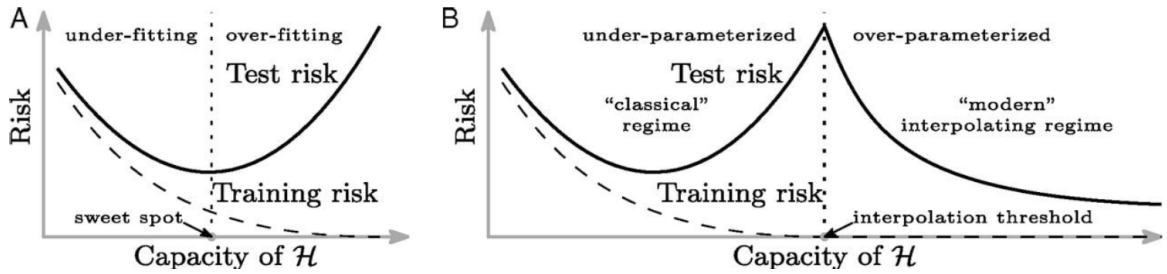



Figure 26: Double descent: test error peaks at the interpolation threshold ($p \approx n$), then **decreases** as $p \gg n$. This contradicts the classical U-shaped bias-variance tradeoff curve.

## 43.1 The Three Regimes

**Double Descent Explained**

The double descent curve has three distinct regions:

1. **Underparameterised regime** ($p < n$): Classical bias-variance tradeoff applies. Test error initially decreases with complexity (reducing bias), then increases (rising variance).
2. **Interpolation threshold** ($p \approx n$): The model can barely fit the training data. It is maximally sensitive to noise—small perturbations in the data cause large changes in the fitted model. Test error peaks here.
3. **Overparameterised regime** ($p \gg n$): Many solutions exist that perfectly interpolate the training data. Among these, the minimum-norm solution (found by gradient descent or SVD) generalises surprisingly well. Test error decreases again.

**Why is the interpolation threshold so bad?** At $p \approx n$, the system $X\beta = y$ is just barely solvable—there is essentially one unique solution that fits the training data perfectly. This solution has no "slack" to avoid fitting noise, and is extremely sensitive to perturbations. It is the worst of both worlds: complex enough to fit noise, but not complex enough to benefit from implicit regularisation.

## 43.2 Explaining the Phenomenon: The Manifold Hypothesis

Why does test error decrease in the overparameterised regime? The **manifold hypothesis** provides key intuition.

**The Manifold Hypothesis**

High-dimensional data often lies on or near a low-dimensional **manifold**—a surface that locally resembles Euclidean space of lower dimension.
**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
- A manifold is a mathematical space that might locally look like flat Euclidean space but has more complex, curved structure globally

**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.

**Examples of Low Intrinsic Dimensionality**

- **Face images**: Live in a space of millions of pixels, but meaningful variations (pose, lighting, expression, identity) span perhaps tens of 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

**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.

## 44 Modern Analysis: The $k$-Split Perspective

To understand benign overfitting rigorously, we decompose the problem using singular value decomposition (SVD). This provides a mathematical framework for separating "signal" from "noise" in high-dimensional data.

## 44.1 The Interpolation Threshold

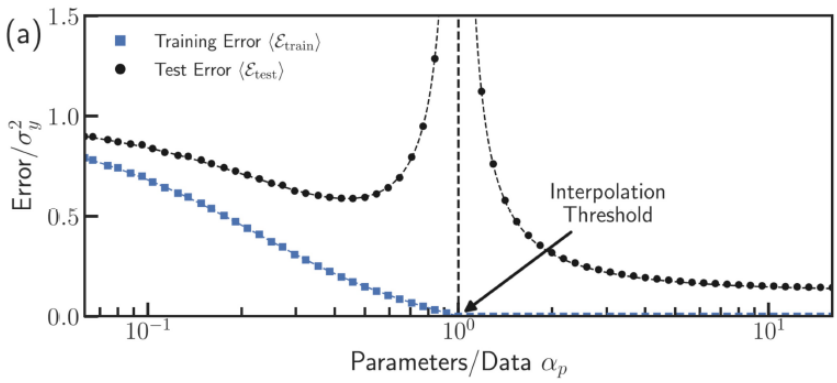



Figure 27: At $p \approx n$, the model achieves perfect interpolation—zero training error. The system $X\beta = y$ transitions from overdetermined (unique solution, typically positive training error) to underdetermined (infinitely many solutions, zero training error).

When $p > n$, the linear system $X\beta = y$ is underdetermined—infinitely many coefficient vectors $\beta$ achieve zero training error. The key question becomes: *which* interpolating solution does our algorithm find, and how well does it generalise?

## 44.2 SVD to the Rescue

When $p > n$, the matrix $X^\top X$ is singular (rank at most $n < p$), so we cannot compute $(X^\top X)^{-1}$ directly. Instead, we use the SVD.

**Minimum-Norm Solution via SVD**

Given the SVD $X = U\Sigma V^\top$ where:

- $U$ is $n \times r$ with orthonormal columns (left singular vectors)
- $\Sigma$ is $r \times r$ diagonal with singular values $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$
- $V$ is $p \times r$ with orthonormal columns (right singular vectors)
- $r = \text{rank}(X) = \min(n, p)$ when $X$ has full rank

The **minimum-norm** OLS solution is:

$$\hat{\beta} = V\Sigma^{-1}U^\top y \tag{32}$$

This is the solution with smallest $\|\beta\|_2$ among all vectors satisfying $X\beta = y$.
**Predictions on training data**:

$$\hat{y} = X\hat{\beta} = U\Sigma V^\top \cdot V\Sigma^{-1}U^\top y = UU^\top y \tag{33}$$

The matrix $UU^\top$ projects onto the column space of $X$.

**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 $\tilde{X}\hat{\beta} = \tilde{X}V\Sigma^{-1}U^\top y$.

### 44.3 Splitting Signal from Noise: The $k$-Split

The key insight of modern benign overfitting theory is to conceptually partition the coefficient vector based on singular value magnitudes:

$$\beta^* = [\beta^*_{1:k}, \beta^*_{k+1:p}] \tag{34}$$

**The $k$-Split Decomposition**

**Low-dimensional part** ($\beta^*_{1:k}$):

- Corresponds to the first $k$ singular vectors (largest singular values)
- Captures the **signal**—the structured, predictive variation in the data
- Behaves like classical low-dimensional OLS: well-estimated, low bias, moderate variance

**High-dimensional part** ($\beta^*_{k+1:p}$):

- Corresponds to the remaining $p - k$ singular vectors (smaller singular values)
- Represents the **noise subspace**—variation that adds complexity without predictive power
- Key question: does this part help or hurt generalisation?

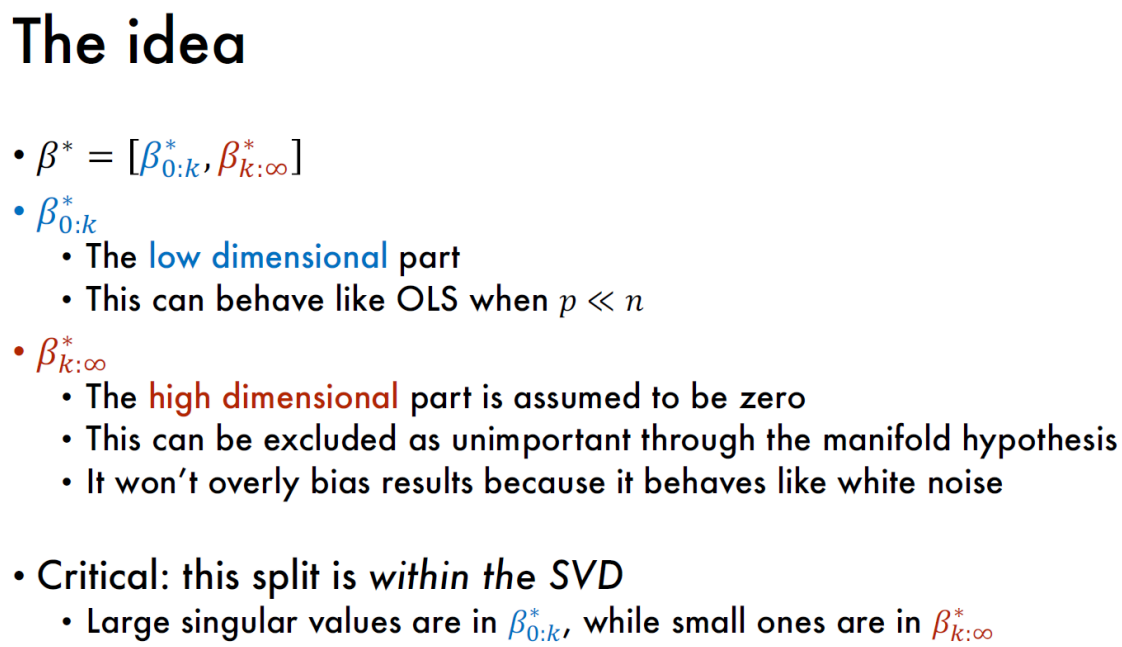



Figure 28: SVD naturally orders dimensions by importance. The first $k$ singular values (left portion) capture structured signal; the remainder (right portion) capture noise-like variation. The split occurs where singular values transition from "large" to "small."

**SVD as Automatic Feature Selection**

The SVD accomplishes several things simultaneously:

1. **Orthogonalisation**: Each transformed feature (singular vector) is orthogonal to all others—no collinearity
2. **Ordering by importance**: Features are automatically ranked by singular value magnitude (variance explained)
3. **Variance concentration**: The first $k$ features capture most of the variance
4. **Rotation without distortion**: 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.

### 44.4 Two Perspectives on the Manifold Hypothesis

The $k$-split provides a formal operationalisation of the manifold hypothesis:

**Formal Interpretations of the Manifold Hypothesis**

**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.

**When does this work?** The $k$-split approach is effective for **highly structured data**:

- The low-dimensional part has high singular values—these dimensions "matter"
- SVD reshuffles the data so that the most important components come first
- After the SVD rotation, each column is independent of all others, and importance is monotonically ordered
- This reveals and exploits intrinsic structure automatically

# 45 Benign Overfitting: When Interpolation Works

"Benign overfitting" describes the counterintuitive scenario where a model that perfectly fits training data (including noise) still generalises well to unseen data. This is not magic—it requires specific conditions.

## 45.1 The Risk Bound

**Risk Bound for Benign Overfitting**

Under appropriate conditions, the excess risk is bounded by:

$$R(\hat{\beta}) - R(\beta^*) \lesssim \frac{\sigma^2}{c}\left(\frac{k^*}{n} + \frac{n}{R_{k^*}(\Sigma)}\right) \tag{35}$$

where:

- $\sigma^2$: Noise variance (irreducible error from the data-generating process)
- $c$: A constant that depends on distributional assumptions (the precise value requires deep mathematical analysis)
- $k^*$: The "split point"—number of dimensions in the signal subspace
- $n$: Sample size
- $R_{k^*}(\Sigma)$: Effective rank of the covariance matrix in the high-dimensional part

## 45.2 Understanding the Two Terms

The bound has two additive components, corresponding to the two parts of the $k$-split:
**First term: $k^*/n$ (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 values) 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: $n/R_{k^*}(\Sigma)$ (high-dimensional contribution)**

- This captures the contribution from the high-dimensional part
- $R_{k^*}(\Sigma)$ measures the "effective number of directions" in the high-dimensional subspace
- It is a technical rank condition on the covariance matrix—specifically, a measure of how spread out the singular values are
- When $R_{k^*}(\Sigma)$ is large, this term is small: high-dimensional noise "averages out"

**What is $R_{k^*}(\Sigma)$?**

The effective rank $R_{k^*}(\Sigma)$ captures the spread of singular values in the high-dimensional part:

- Each SVD dimension is orthogonal to all others
- High $R_{k^*}(\Sigma)$ means the high-dimensional part "moves in many different directions"
- This is precisely the "white noise" assumption: variation is spread across many dimensions rather than concentrated
- If the noise dimensions point in many different directions, they average out rather than systematically biasing predictions
- With high effective rank, we can effectively estimate zero for coefficients in this part—the noise cancels

## 45.3 When is Overfitting Benign?

**Three Conditions for Benign Overfitting**

Benign overfitting occurs when:

1. **Low intrinsic dimension**: The signal lives in a low-dimensional subspace ($k^* \ll p$). Most of the predictive information is captured by a small number of dimensions.
2. **White noise in high dimensions**: The remaining dimensions behave like isotropic (direction-independent) noise. No preferred directions, no structure to exploit or be misled by.
3. **Sufficient spread**: The noise spreads across *many* directions ($R_{k^*}(\Sigma)$ is large). This ensures averaging effects dominate.

**When these hold**: Generalisation in the interpolating regime is "as if" you ran OLS on just the $k^*$ intrinsic dimensions. The high-dimensional noise part contributes negligibly.

**NB!**

**Critical assumption**: The high-dimensional part must behave like white noise for benign overfitting to work.
If the high-dimensional part has structure (correlations between dimensions, preferred directions, systematic patterns), it can bias the model and hurt generalisation. The theory does *not* say interpolation is always benign—only that it can be benign under specific conditions.

## 45.4 Why SVD Makes This Automatic

When you fit OLS using SVD (which is necessary when $p > n$), the decomposition automatically:

1. **Identifies signal directions**: The SVD finds directions of maximum variance, which typically correspond to signal
2. **Orders by importance**: Dimensions are ranked by singular value magnitude

3. **Finds minimum-norm solution**: Among all interpolating solutions, SVD gives the one with smallest $\|\beta\|_2$
4. **Avoids overweighting noise**: The minimum-norm constraint acts as **implicit regularisation**, preventing the solution from amplifying noise directions

The minimum-norm property is crucial: among infinitely many solutions that fit the training data perfectly, the SVD-based solution is "smoothest" in the sense of having smallest coefficient norm. This is a form of Occam's razor built into the mathematics.

**Implicit Regularisation via Minimum Norm**

The minimum-norm OLS solution from SVD provides a double benefit:

1. **Enables computation**: Bypasses the singularity of $X^\top X$ when $p > n$
2. **Provides regularisation**: Among all perfect-fit solutions, selects the "simplest" one

This is why gradient descent on overparameterised models often works well: it tends to find minimum-norm solutions, which have this implicit regularisation property.

## 46 Implications for High-Dimensional Models

**Why High-Dimensional Models Can Generalise**

As long as the high-dimensional part of $X$ covers a wide array of directions, the risk of overfitting—though present—does not necessarily impair generalisation. The mechanism:

- Diversity in noise directions distributes model complexity across many dimensions
- No single noise direction dominates or systematically biases predictions
- The minimum-norm solution naturally de-emphasises noise directions (they have small coefficients)
- The model effectively "averages over" the noise, leaving only the signal

This provides a resolution to the apparent paradox of deep learning success:

**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

### 46.1 The Importance of Data Structure

Benign overfitting underscores a fundamental insight: **data structure matters more than nominal dimensionality**.

- The effective complexity of learning depends on intrinsic, not ambient, dimensionality
- Models can be "complex" in parameter count but "simple" in what they actually learn
- Understanding the geometry and distribution of the feature space is crucial for high-dimensional analysis
- Not all high-dimensional problems are equally hard—structure makes them tractable

## 47 Practical Implications

**Key Takeaways for Practice**

1. **Interpolation is not always bad**: In high dimensions with structured data, fitting training data perfectly can still yield good generalisation
2. **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. Beware of systematic noise patterns.
4. **Implicit regularisation is powerful**: Minimum-norm solutions (from SVD, gradient descent) automatically avoid overfitting to noise in many cases
5. **Do not fear overparameterisation**: Modern deep learning operates successfully in this regime. The interpolation threshold ($p \approx n$) is dangerous, but $p \gg n$ can be safe.
6. **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.**
It requires:

- Data with genuine low-dimensional structure
- High-dimensional noise that is approximately isotropic
- Appropriate inductive biases (minimum-norm, early stopping, architecture choices)
- Being sufficiently past the interpolation threshold ($p \gg n$, not $p \approx n$)

Without these conditions, overfitting remains harmful. 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

# 48 Summary

**Key Concepts from Week 5**

1. **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
2. **Double descent**: Test error peaks at the interpolation threshold ($p \approx n$), then *decreases* in the overparameterised regime
   - Contradicts classical U-shaped bias-variance curves
   - Explained by implicit regularisation in overparameterised models
3. **Manifold hypothesis**: High-dimensional data often has low intrinsic dimensionality
   - Real data lives near low-dimensional manifolds
   - This structure enables learning despite high ambient dimensionality
4. **$k$-split analysis**: SVD separates signal (large singular values) from noise (small singular values)
   - Low-dimensional part behaves like classical regression
   - High-dimensional part averages out if noise is isotropic
5. **Benign overfitting conditions**: Interpolating models generalise well when:
   - Signal has low intrinsic dimension
   - Noise is white (isotropic, spread across many directions)
   - Minimum-norm solutions are used (implicit regularisation)
6. **Implicit regularisation**: Minimum-norm solutions from SVD/gradient descent avoid amplifying noise directions
   - This happens automatically, not through explicit penalties
   - Explains why overparameterised models can generalise

# 49 Motivation: A New Way to Think About Similarity

Kernels offer a fundamentally different perspective on machine learning: instead of thinking about models as weighted combinations of *features*, we think about them as weighted combinations of *observations*. This shift in viewpoint leads to remarkably flexible models that can capture complex non-linear relationships.

**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.

# 50 An Alternative View of Regression

## 50.1 Two Equivalent Formulations of Ridge Regression

Traditional regression assigns coefficients to *features*—we compute a linear combination of features, weighted by parameter coefficients. An alternative perspective assigns weights to *observations* based on their similarity to the point of interest.
Consider ridge regression. Using the matrix identity $(A + \lambda I)^{-1}A = A(A + \lambda I)^{-1}$, we can write the fitted values in two equivalent ways:

**Two Views of Ridge Regression**

**Feature-space view** (traditional):

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

**Observation-space view** (kernel):

$$\hat{y} = \underbrace{XX^\top}_{n\times n}(XX^\top + \lambda I_n)^{-1}y$$

Both formulations give identical predictions, but they differ in computational cost:

- The feature-space view inverts a $p \times p$ matrix
- The observation-space view inverts an $n \times n$ matrix

**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)

After feature expansion, $p$ can become very large (even infinite), making the observation-space view essential.

## 50.2 What Do These Matrices Represent?

The matrices $X^\top X$ and $XX^\top$ have natural interpretations in terms of similarity:

- $X^\top X$ is a $p \times p$ matrix: each element $(X^\top X)_{jk}$ is the dot product between feature columns $j$ and $k$. This captures **similarity between features**.
- $XX^\top$ is an $n \times n$ matrix: each element $(XX^\top)_{ij}$ is the dot product between observation rows $i$ and $j$. This captures **similarity between observations**.

Understanding why dot products measure similarity is crucial for kernel methods.

## 50.3 Similarity as Dot Product

**Dot Product and Distance**

The squared Euclidean distance between two vectors can be expressed in terms of dot products:

$$\begin{aligned} d^2(x, x') = \|x - x'\|^2 &= \sum_{i=1}^{d} (x_i - x_i')^2 \\ &= \sum_{i=1}^{d} x_i^2 - 2\sum_{i=1}^{d} x_i x_i' + \sum_{i=1}^{d} x_i'^2 \\ &= x^\top x - 2x^\top x' + x'^\top x' \end{aligned}$$

For normalised vectors ($\|x\| = \|x'\| = 1$):

$$d^2(x, x') = 2(1 - x^\top x')$$

Thus: **dot product** $\propto$ **similarity** $\propto 1-$ **distance**.

The dot product has a beautiful geometric interpretation:

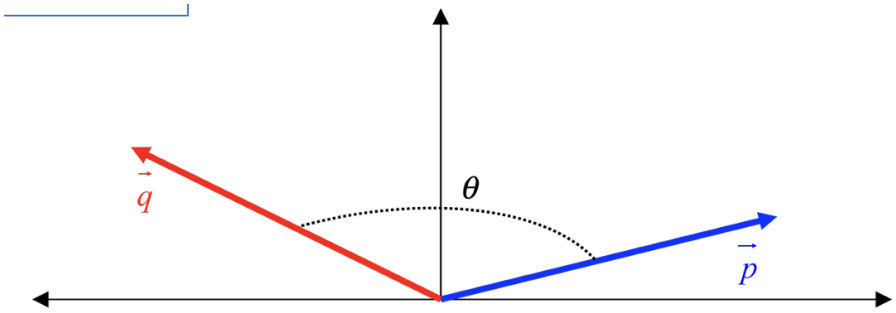



Figure 29: The dot product $x \cdot y = \|x\|\|y\| \cos\theta$ captures both magnitude and direction. When $\theta$ is small (similar directions), $\cos\theta$ is large and the dot product is large.

**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**

- $\theta = 0°$ (parallel): $\cos\theta = 1 \Rightarrow$ maximum similarity
- $\theta = 90°$ (orthogonal): $\cos\theta = 0 \Rightarrow$ no similarity (no overlap in any dimension)
- $\theta = 180°$ (anti-parallel): $\cos\theta = -1 \Rightarrow$ maximum dissimilarity

**Key intuition**: If similar, angle is small $\rightarrow$ cosine is large $\rightarrow$ dot product is large.

## 50.4 Regression as Similarity-Weighted Averaging

In the observation-space view, prediction becomes a weighted average of training labels, where the weights reflect similarity:

**Prediction as Similarity-Weighted Average**

For a test point $\tilde{x}$, we can write the ridge regression prediction as:

$$\hat{y}(\tilde{x}) = \tilde{x}X^\top(XX^\top + \lambda I_n)^{-1}y = \sum_{i=1}^{n} w_i y_i$$

where the weights are:

$$w = \tilde{x}X^\top(XX^\top + \lambda I_n)^{-1}$$

The term $\tilde{x}X^\top$ computes the similarity between the test point $\tilde{x}$ and each training point. Observations more similar to $\tilde{x}$ receive higher weights in the prediction.

This is a powerful reframing: 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 $\lambda$ prevents overfitting by smoothing the weights.

**NB!**

In linear regression (and ridge regression), the weights evolve **linearly** with distance. This is often too inflexible—we may want nearby points to have much higher weight than distant ones. 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 **non-linear similarity measures**—kernels.

### 50.5 The Importance of Defining Similarity Correctly

Different notions of similarity lead to fundamentally different models. The figure below shows datasets with identical linear correlations but vastly different structures:

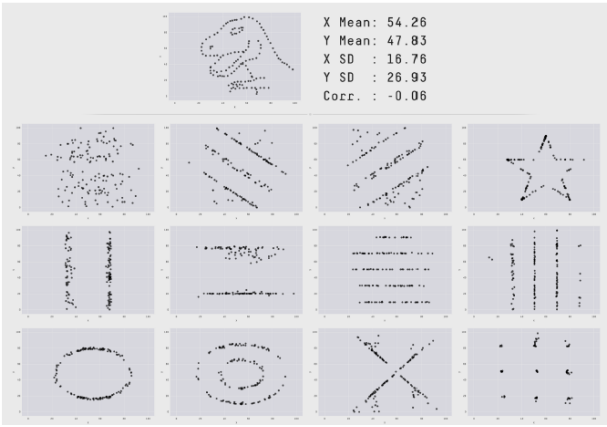



Figure 30: All four datasets have the same correlation coefficient, but their structures differ dramatically. Linear correlation (based on Euclidean distance) cannot distinguish them.

**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.

## 51 Feature Expansion and the Kernel Trick

### 51.1 Feature Expansion

To capture non-linear relationships with linear models, we transform the input space:

**Feature Expansion**

A **feature map** $\phi : \mathbb{R}^d \to \mathbb{R}^D$ transforms inputs into a (typically higher-dimensional) space.
**Examples:**

- **Polynomial**: $\phi(x) = [1, x, x^2, \ldots, x^M]$ for degree $M$
- **Trigonometric**: $\phi(x) = [1, \cos(x), \cos(2x), \ldots, \cos(Mx)]$

In the expanded space, a linear model can capture non-linear relationships in the original space.

We have used feature expansion throughout this course—polynomial regression and Fourier basis expansions are examples. The key insight is that linear regression in a transformed feature space is equivalent to non-linear regression in the original space.
**Example**: For a 2D input $x = [x_1, x_2]$, a polynomial expansion might be:

$$\phi(x) = [x_1, x_2, x_1^2, x_2^2, x_1 x_2]$$

This maps 2 features to 5 features, enabling the model to capture quadratic relationships.

## 51.2 The Computational Problem

As the degree of expansion $M$ increases, the dimensionality of $\phi(x)$ grows rapidly. For a polynomial of degree $M$ in $d$ dimensions, the number of features is $\binom{d+M}{M}$, which grows combinatorially. For trigonometric expansions, as the frequency increases, so does dimensionality. And for some applications, we want $M \to \infty$—an infinite-dimensional feature space!
The problem: computing $\phi(x)^\top \phi(x')$ explicitly becomes intractable when $\phi$ is very high- or infinite-dimensional.

## 51.3 The Kernel Trick

**Kernel Definition**

A **kernel** is a function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ that computes the dot product in feature space without explicitly constructing the feature vectors:

$$k(x, x') = \phi(x)^\top \phi(x')$$

The **kernel trick**: replace all occurrences of $x^\top x'$ with $k(x, x')$.

**Why Kernels Work**

Many machine learning algorithms (ridge regression, SVMs, PCA) depend on the data only through dot products $x_i^\top x_j$.
If we can compute $k(x_i, x_j) = \phi(x_i)^\top \phi(x_j)$ *without* computing $\phi$ explicitly, we get the benefits of high-dimensional (even infinite-dimensional) feature spaces at low computational cost.
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

### 51.4 The Gram Matrix

**Gram Matrix**

The **Gram matrix** (or kernel matrix) $K$ is an $n \times n$ matrix containing all pairwise kernel evaluations:

$$K_{ij} = k(x_i, x_j) = \phi(x_i)^\top \phi(x_j)$$

A valid kernel produces a **positive semi-definite** Gram matrix, meaning all eigenvalues are $\geq 0$. This is equivalent to requiring that the kernel "looks like" a dot product—it must arise from some (possibly implicit) feature map.

## 52 Common Kernels

### 52.1 Polynomial Kernel

**Polynomial Kernel**

$$k(x, x') = (x^\top x' + c)^M$$

Parameters:

- $M$: degree of the polynomial
- $c$: constant term (controls influence of lower-degree terms)

**Worked example**: Let $M = 2$, $c = 0$, and $x, z \in \mathbb{R}^2$:

$$\begin{aligned} k(x, z) &= (x^\top z)^2 = (x_1 z_1 + x_2 z_2)^2 \\ &= x_1^2 z_1^2 + 2 x_1 x_2 z_1 z_2 + x_2^2 z_2^2 \\ &= \underbrace{(x_1^2, \sqrt{2} x_1 x_2, x_2^2)}_{\phi(x)} \cdot \underbrace{(z_1^2, \sqrt{2} z_1 z_2, z_2^2)}_{\phi(z)} \\ &= \phi(x)^\top \phi(z) \end{aligned}$$

The kernel computes this 3-dimensional dot product directly from the 2-dimensional inputs, without explicitly constructing $\phi(x)$ and $\phi(z)$. For higher degrees, the savings become dramatic: a degree-10 polynomial in 100 dimensions would require computing features in a space of dimension $\binom{110}{10} \approx 10^{13}$.

**Polynomial Kernel: Key Takeaway**

The polynomial kernel implicitly computes dot products in a space containing all polynomial terms up to degree $M$. This allows linear algorithms to capture polynomial relationships without the computational burden of explicit feature expansion.

## 52.2 Gaussian (RBF) Kernel

The Gaussian kernel, also called the Radial Basis Function (RBF) kernel, is perhaps the most widely used kernel:

**Gaussian/RBF Kernel**

$$k(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right)$$

Parameters:

- $\sigma$ (bandwidth): controls the "width" of the kernel
- Small $\sigma$: only very close points are considered similar
- Large $\sigma$: distant points retain some similarity

The formula has an intuitive interpretation:

1. Take the difference between $x$ and $x'$
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 $x = x'$ and decays smoothly towards 0 as points become distant.

### 52.2.1 The RBF Kernel is Infinite-Dimensional

**Expanding the Gaussian Kernel**

To understand the feature space, expand the squared distance:

$$\begin{aligned} k(x, x') &= \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right) \\ &= \exp\left(-\frac{x^\top x}{2\sigma^2}\right) \exp\left(\frac{x^\top x'}{\sigma^2}\right) \exp\left(-\frac{x'^\top x'}{2\sigma^2}\right) \end{aligned}$$

The middle term contains the interaction between $x$ and $x'$. The outer terms are normalisation factors depending only on individual vectors.

The key insight comes from the Taylor expansion of the exponential:

$$e^z = \sum_{k=0}^{\infty} \frac{z^k}{k!} = 1 + z + \frac{z^2}{2!} + \frac{z^3}{3!} + \cdots$$

**The RBF Kernel is Infinite-Dimensional**

Applying the Taylor expansion to $\exp(x^\top x'/\sigma^2)$ shows that the RBF kernel corresponds to an **infinite-dimensional** feature space containing:

- All polynomial terms of all degrees
- Each term weighted by $1/k!$, which decreases rapidly with degree

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.

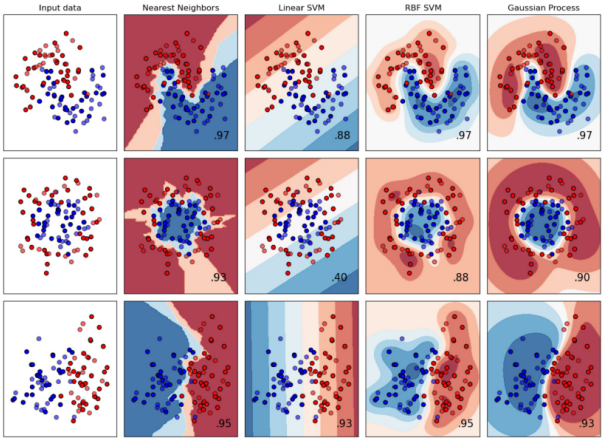



Figure 31: Left: Linear regression (rigid, global). Right: RBF kernel regression allows locally-weighted similarity, adapting flexibly to the data structure.

### 52.3 Other Common Kernels

Beyond polynomial and RBF kernels, many other kernels exist for specific applications:

- **Periodic kernel**: For cyclical/seasonal patterns
- **String kernels**: For text and sequence data
- **Graph kernels**: For structured/relational data

The choice of kernel encodes your beliefs about the structure of similarity in your problem.

## 53 Combining Kernels

One of the most powerful aspects of kernel methods is that kernels can be combined to express complex notions of similarity.

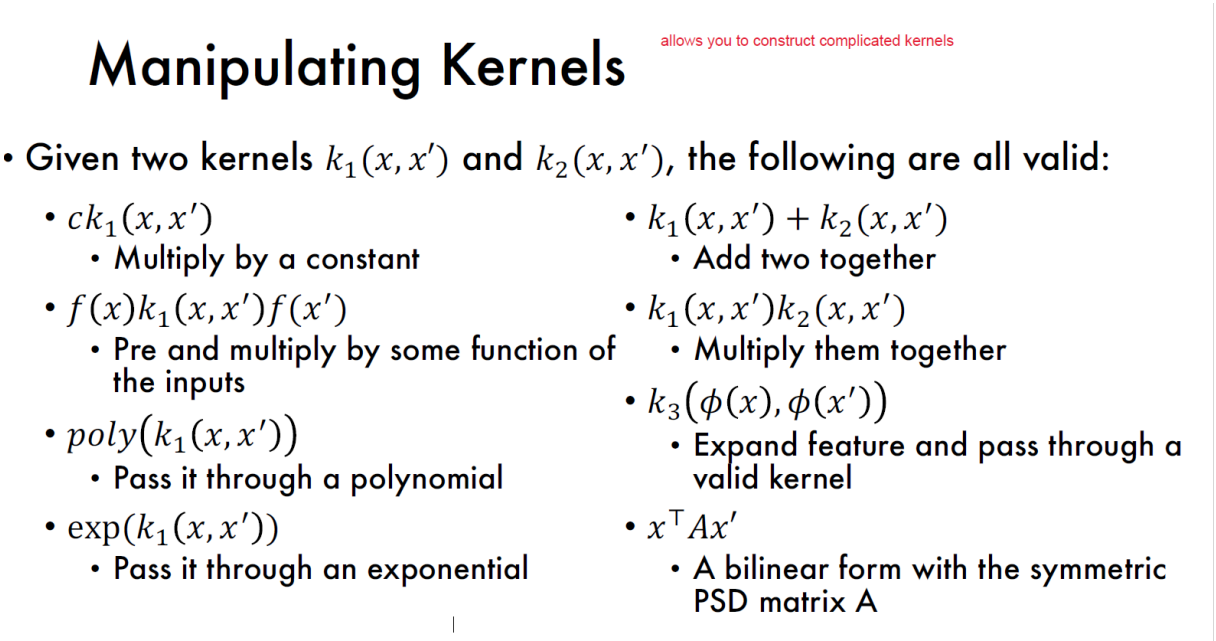



Figure 32: Rules for constructing valid kernels from simpler ones. Any combination that preserves positive semi-definiteness yields a valid kernel.

**Kernel Combination Rules**

If $k_1$ and $k_2$ are valid kernels, then so are:

- $\alpha k_1 + \beta k_2$ for $\alpha, \beta \geq 0$ (weighted sum)
- $k_1 \cdot k_2$ (product)
- $f(x)k_1(x, x')f(x')$ for any function $f$
- $\exp(k_1)$ (exponential of a kernel)

**Building Custom Similarity Measures**

These rules allow constructing sophisticated kernels that encode domain knowledge:

$$k = \alpha \cdot k_{\text{local}} + (1 - \alpha) \cdot k_{\text{global}}$$

For example, combining:

- A narrow Gaussian kernel (capturing local patterns)
- A wide Gaussian kernel (capturing global trends)
- A periodic kernel (capturing cyclical behaviour)

The hyperparameter $\alpha$ balances the contributions of different similarity notions.

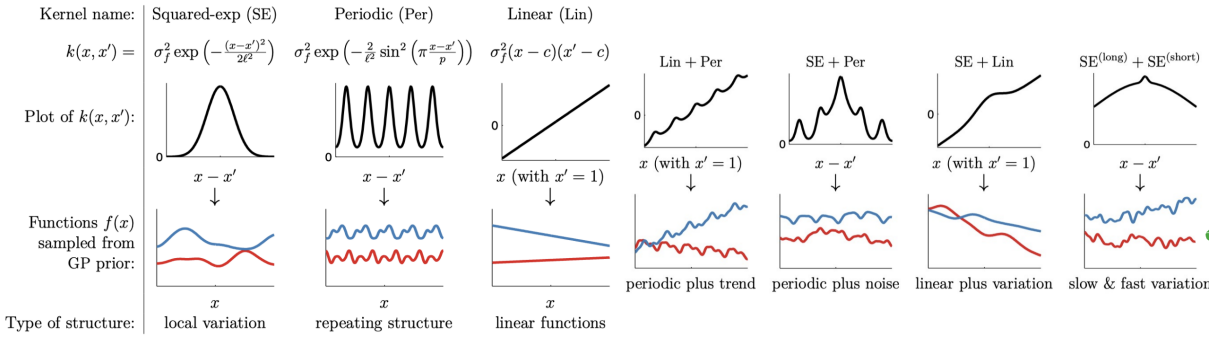



Figure 33: Combining kernels: Gaussian (local), periodic (cyclical), and mixed. Adjusting $\sigma^2$ combines wide global structure with high-frequency local variation.

- **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.

# 54 Kernel Methods in Practice

## 54.1 Kernel Ridge Regression

Replacing dot products with kernel evaluations transforms ridge regression into a powerful non-linear method:

**Kernel Ridge Regression**

Starting from the observation-space view of ridge regression:

$$\hat{y} = \tilde{X}X^\top(XX^\top + \lambda I_n)^{-1}y$$

Replace $XX^\top$ with kernel matrix $K$ and $\tilde{X}X^\top$ with kernel evaluations:

$$\hat{y} = K_{\tilde{x}X}(K_{XX} + \lambda I_n)^{-1}y$$

where:

- $K_{XX}$ is the $n \times n$ Gram matrix of training points
- $K_{\tilde{x}X}$ contains kernel evaluations between test point(s) and training points

**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
- **Computational cost**: $O(n^3)$ for matrix inversion—cubic in the number of training points

**Hyperparameters to tune:**

- Choice of kernel (and its parameters, e.g., $\sigma$ for RBF)
- Regularisation parameter $\lambda$

As flexibility increases, careful tuning becomes increasingly important.

## 54.2 K-Nearest Neighbours (KNN)

An alternative to global kernel methods is to use only local information:

**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 to $\tilde{x}$.

**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)

**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.

## 54.3 Comparing Global and Local Methods

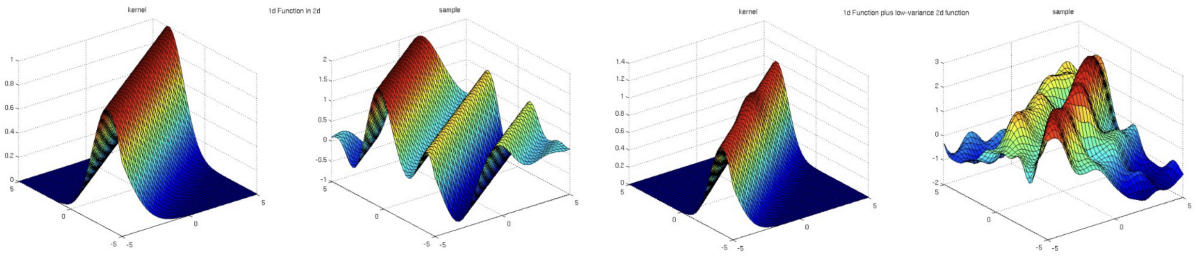


Figure 34: Kernels can induce low-rank global structure (capturing overall trends) while allowing for local variation (adapting to fine-grained patterns). This balances smoothness with flexibility.

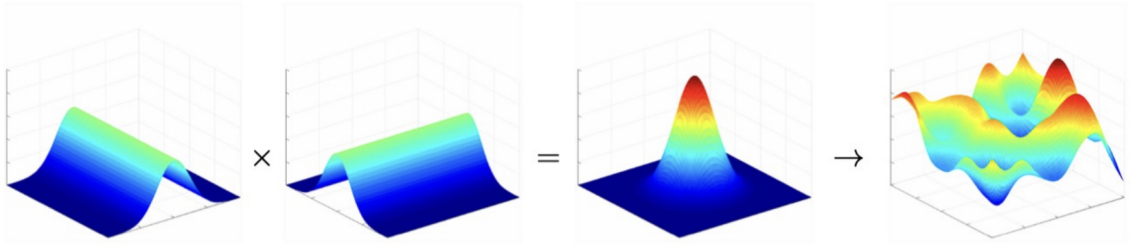



Figure 35: Different features may demand different notions of similarity. By combining or learning kernel parameters, models can discover which features (or combinations) are most indicative of similarity.

**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.

## 55 The Curse of Dimensionality

Kernel methods rely fundamentally on meaningful notions of distance. In high-dimensional spaces, this foundation crumbles.

**NB!**

**The curse of dimensionality**: In high dimensions, **distance becomes meaningless**—all points become approximately equidistant from each other.

### 55.1 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 the hypercube: $2^d$
- Volume of a ball of radius $\epsilon$: $V_d(\epsilon) = \frac{\pi^{d/2}}{\Gamma(d/2+1)} \epsilon^d$

As $d$ increases, the fraction of the hypercube's volume contained in the $\epsilon$-ball shrinks exponentially.

What does this mean practically? To capture a fixed fraction of data points as "neighbours," we need $\epsilon$ to grow dramatically with dimension:

| d | Volume of domain of $X$ | Volume of sphere | Fraction of data nearby | $\epsilon = \frac{1}{2}$ |
|---|---|---|---|---|
| 1 | $2^1$ | $2\epsilon$ | $\epsilon$ | $\frac{1}{2} = 0.5$ |
| 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 36: To capture 10% of data as $d$ increases, the neighbourhood radius $\epsilon$ must grow toward the boundary of the space.

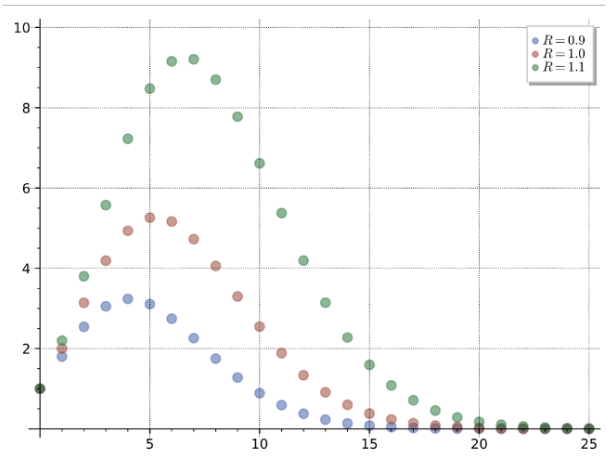



Figure 37: Volume concentration in high dimensions: most volume lies near the boundary, and the concept of "local neighbourhood" becomes meaningless.

## 55.2 Implications for Machine Learning

**Consequences of the Curse**

1. **Local methods become global**: In high dimensions, nothing is "close"—the amount of data in any local neighbourhood shrinks toward zero
2. **Extrapolation replaces interpolation**: Predictions at new points are no longer informed by "nearby" training examples
3. **Distance metrics lose discriminative power**: All points become approximately equidistant

**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)

**NB!**

**When NOT to use kernel methods:**

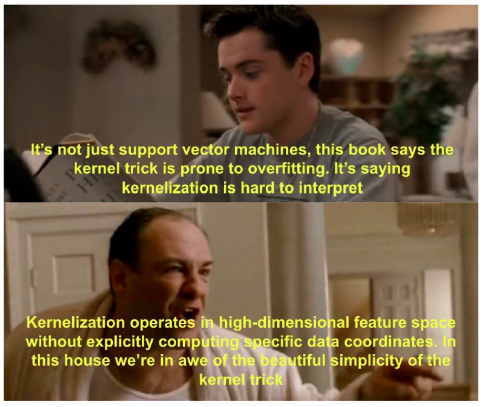
- **High-dimensional features**: Distance becomes meaningless; everything is far apart
- **Small sample size**: Need $O(n^2)$ kernel evaluations and $O(n^3)$ for inversion
- **Large datasets**: The $n \times n$ kernel matrix becomes prohibitively large

Kernel methods shine with moderate-sized datasets in low-to-moderate dimensions, where domain knowledge can inform kernel choice.

# 56 Summary

**Key Concepts from Week 6a**

1. **Dual view of regression**: We can weight observations by similarity, not just features by coefficients
2. **Kernels as similarity**: A kernel $k(x, x') = \phi(x)^\top \phi(x')$ computes dot products in (potentially infinite-dimensional) feature spaces
3. **The kernel trick**: Replace $x^\top x'$ with $k(x, x')$ to work in high-dimensional spaces without explicit computation
4. **RBF kernel**: Infinite-dimensional, contains all polynomial terms, prefers smooth functions via $1/k!$ weighting
5. **Combining kernels**: Build custom similarity measures encoding domain knowledge about local vs. global structure, periodicity, feature importance
6. **Curse of dimensionality**: Distance loses meaning in high dimensions; kernel methods require careful application



## Overview

This week introduces the qualitative dimensions of fairness in machine learning—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.
**Key themes:**

- How ML systems amplify historical biases through feedback loops
- The distinction between allocative and representational harm
- Three dimensions of fairness: legitimacy, relative treatment, and procedural fairness
- Different types of automation and their distinct fairness challenges
- Sources of unfairness in data-driven systems
- The importance of agency, recourse, and accountability

# 57 Machine Learning in Context

ML models learn patterns from data. When that data reflects historical biases—as virtually all real-world data does—models can **amplify** those biases through automation. This is not a bug in the algorithm; it is a consequence of the fundamental premise that models learn to replicate patterns they observe.

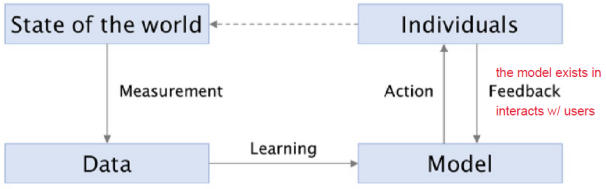



Figure 38: ML systems exist within broader social and institutional contexts. The model is just one component in a sociotechnical system that includes data collection, deployment decisions, and feedback loops.

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.

## 57.1 Encoded Bias: Which Patterns Should We Replicate?


Figure 39: Language models encode societal biases. This example from Google Translate (Turkish to English) shows how gender-neutral Turkish pronouns are translated using stereotypical gender associations—"he" for doctor/soldier, "she" for nurse/teacher. Which patterns should we replicate, and which not?

The Turkish language 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. But should it?
This raises a fundamental question that pervades ML fairness: **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.

## 57.2 Feedback Loops: Predictions as Self-Fulfilling Prophecies

**NB!**

**Feedback loops**: Predictions can become self-fulfilling prophecies, creating cycles where biased predictions generate biased data that reinforces the original bias.
**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*.

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

**Construct Validity: What Are We Actually Measuring?**

From a social science perspective, this is a **construct validity** problem. The model's target variable (arrests) is a proxy for the construct we actually care about (crime). But:

- Arrests $\neq$ Crimes committed
- Arrests = f(Crimes committed, Police presence, Prosecution decisions, ...)

The proxy (arrests) conflates the underlying construct (crime) with the measurement process (policing). When we optimise for the proxy, we may be optimising for something quite different from what we intended.
This is a pervasive issue in ML: we rarely have direct access to the quantities we care about, so we train on proxies. Understanding the gap between proxy and construct is essential for evaluating whether a model is fair.

## 58 Types of Harm

Not all harms from ML systems are alike. A foundational distinction separates two broad categories:

**Allocative vs Representational Harm**

**Allocative harm**: A system withholds a resource or opportunity based on group membership.

- **Examples**: Loan denials, hiring decisions, benefit eligibility, bail decisions, housing applications
- **Characteristics**: Direct, measurable impact on individuals; involves tangible resources or opportunities
- **Detection**: Relatively easier to measure through outcome disparities
- **Remediation**: Can potentially be addressed through policy, quotas, or algorithmic constraints

**Representational harm**: A system reinforces stereotypes or subordination of groups along the lines of identity.

- **Examples**: Search result rankings, image captioning, language generation, recommendation systems, voice assistants
- **Characteristics**: Shapes perceptions and reinforces social hierarchies; harder to quantify
- **Detection**: Often requires qualitative analysis, user studies, or careful auditing
- **Remediation**: May require changes to training data, model architecture, or post-processing

**Why the distinction matters**: These two types of harm require different analytical frameworks and different interventions. Allocative harm is often amenable to the quantitative fairness metrics we will develop in Week 7. Representational harm is more subtle and may resist quantification—but it can be equally or more damaging in aggregate.

**Cascading Effects**

Representational harm can lead to allocative harm through indirect channels:

- Stereotypical image search results shape employer perceptions
- Biased language models influence hiring algorithms
- Underrepresentation in training data leads to worse performance for minority groups

The two categories interact: representational harm creates the conditions for allocative harm, while allocative harm generates the disparate outcomes that become training data for future representational harm.

# 59 What is Fairness?

"Fairness" is not a single concept but a family of related concerns. 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
   - Some applications may be fundamentally illegitimate regardless of how "fair" their implementation
   - E.g., predicting criminality from facial features—is there *any* legitimate use case?
2. **Relative treatment**: How does the system allocate resources across groups?
   - Can be measured rigorously and quantitatively (Week 7)
   - Involves concepts like demographic parity, equalised odds, calibration
   - Different metrics capture different notions of fairness—and they conflict
3. **Procedural fairness**: Is the decision-making process transparent and rational?
   - Especially important for complex models where reasoning is opaque
   - Concerns: explainability, right to reasons, contestability
   - Even a "fair" outcome may be illegitimate if the process is opaque

## 59.1 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.

### 59.2 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

### 59.3 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. Procedural fairness concerns include:

**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.

## 60 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 |

# 61 Problems in Data-Driven Systems

Type 3 automation—learning rules from data—is the most prevalent in modern ML and the most fraught with fairness challenges. Four major sources of unfairness arise:

**Four Sources of Unfairness in Type 3 Automation**

1. **Biased training data**
   - Historical discrimination is encoded in outcomes (who got hired, who got loans, who was arrested)
   - For many sensitive domains, *no unbiased data exists*—the historical record is the biased record
   - Even "ground truth" labels may reflect biased decisions (e.g., performance reviews, medical diagnoses)
   - The model cannot learn to be fairer than its training data without explicit intervention
2. **Proxy mismatch** (construct validity)
   - The target variable does not measure what we actually care about
   - Data is often chosen based on convenience or what powerful actors chose to collect
   - **Example**: US healthcare algorithm used predicted *costs* as a proxy for healthcare *need*
   - **Result**: Systematically under-served patients who could not afford care (the most vulnerable) because their historical costs were low—not because their needs were low
   - The proxy (costs) conflated need with ability to pay
3. **Feature omission**
   - Failing to measure relevant differences between individuals
   - Model treats different people as identical because distinguishing features were not collected
   - What is not measured cannot influence the model's predictions
   - **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
4. **Distribution shift**
   - Training data differs systematically from deployment population
   - **Example**: Medical trials predominantly enrolled white, male, educated, middle-class participants
   - Model may not generalise to underrepresented groups
   - Even a model that is "fair" on training data may be unfair in deployment if the populations differ

### 61.1 The Healthcare Algorithm Example

The US healthcare algorithm case (Obermeyer et al., 2019) is worth examining in detail because it illustrates how proxy mismatch creates unfairness even with "good intentions."

**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).

## 62 Agency, Recourse, and Culpability

A critical dimension of fairness concerns what individuals can *do* about decisions that affect them. This involves questions of agency (can individuals change their outcomes?), recourse (can they understand and act on decisions?), and culpability (who is responsible when things go wrong?).

**Immutable vs Mutable Characteristics**

**Models on immutable characteristics** (age, race, birthplace, genetic markers):

- Individuals cannot change these features
- Their fate is determined by factors outside their control
- Raises fundamental questions of fairness and human dignity
- Using such features may be legally prohibited in many contexts (but models may learn proxies)

**Models on mutable characteristics** (education, employment history, behaviour):

- Individuals could in principle change these features
- If the model uses mutable features, there is an obligation to inform individuals how to improve their outcomes
- **But**: This creates opportunity to "game" the system
- Goodhart's Law applies: "When a measure becomes a target, it ceases to be a good measure"

## 62.1 The Problem of Recourse

**NB!**

**Recourse**: Can individuals understand decisions and act on them?
If a model denies someone a loan, they should know:

- Why they were denied (which factors contributed negatively?)
- What they could change to be approved (actionable feedback)
- Whether the denial was based on legitimate factors (contestability)

Black-box models make recourse difficult or impossible. Even if you can explain *what* features mattered, you may not be able to explain *why* they matter or *how* to change them.
**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%")

## 62.2 Culpability: Who Is Responsible?

When an automated system makes a harmful decision, who is responsible?

**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.
**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.

## 63 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

## 64 Looking Ahead: Quantitative Fairness

In Week 7, we will develop quantitative frameworks for measuring fairness. We will see:

- 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
- 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.

# 65 Summary

**Key Concepts from Week 6b**

1. **Bias amplification**: ML models learn from data that reflects historical discrimination; automation can amplify these biases rather than correcting them
2. **Feedback loops**: Predictions can become self-fulfilling prophecies, creating cycles where biased predictions generate biased data (e.g., predictive policing)
3. **Construct validity**: Proxy variables (arrests, costs) often conflate what we care about with the measurement process; optimising for proxies may optimise for the wrong thing
4. **Allocative vs representational harm**: Resource denial (loans, jobs) versus stereotype reinforcement (search results, language models)—both matter but require different analysis
5. **Three fairness dimensions**:
   - Legitimacy: Should this system exist?
   - Relative treatment: How does it allocate across groups?
   - Procedural fairness: Is the process transparent and contestable?
6. **Three automation types**:
   - Type 1 (explicit rules): Loses human discretion
   - Type 2 (informal judgements): May learn different reasoning
   - Type 3 (learning from data): Inherits all data biases
7. **Four data problems**: Biased training data, proxy mismatch, feature omission, distribution shift
8. **Agency and recourse**: Individuals should understand decisions and have paths to different outcomes; black-box models make this difficult
9. **Culpability**: Responsibility for algorithmic harms is diffused across data collectors, developers, deployers, and operators
10. **Limits of technical solutions**: Fairness is fundamentally a sociotechnical challenge; technical tools can support but not replace human value judgements

# 66 Classification and Risk Scores

Binary classification learns a function $f : \mathcal{X} \to \{0, 1\}$ 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.

## 66.1 Risk Scores and Estimation

**Risk Scores**

A **risk score** $r(x)$ estimates the probability of the positive class:

$$r(x) \approx \mathbb{E}[Y \mid X = x] = P(Y = 1 \mid X = x) \tag{36}$$

This expectation is the probability of the positive class conditioned on the features. Classification is regression "hiding" under the hood—logistic regression estimates $r(x)$, then we threshold to classify:

$$\hat{y}(x) = \mathbf{1}[r(x) > \tau] \tag{37}$$

where $\tau$ is the decision threshold (often 0.5) and $\mathbf{1}[\cdot]$ is the indicator function.

## 66.2 Ideal Model versus Reality

In an **ideal scenario** with perfect knowledge, we could define our classifier's action as:

$$\hat{y}(x) = \mathbf{1}[\mathbb{E}[Y \mid X = x] > 0.5] = \begin{cases} 1 & \text{if } \mathbb{E}[Y \mid X = x] > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

If the expected probability of $Y = 1$ given $X = x$ exceeds 0.5, we predict the positive class; otherwise, we predict 0.

**The reality** is that we don't know $\mathbb{E}[Y \mid X = x]$ *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 $\hat{r}(x) \approx \mathbb{E}[Y \mid X = x]$.

**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.

# 67 Evaluating Classifiers

## 67.1 Accuracy and Its Limitations

Accuracy is the most intuitive metric for classification, defined as the ratio of correct predictions to total predictions:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{1}{n}(TP + TN) \tag{38}$$

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

**How accurate is accurate enough?** The required level of accuracy depends entirely on the application and its implications.

**NB!**

Accuracy treats all errors as equally costly—it makes the implicit assumption that Type I and Type II errors are equivalently bad. In practice, this is rarely true:

- **Medical diagnosis**: False negatives (missed disease) may be catastrophic—a patient goes untreated
- **Spam filtering**: False positives (blocking legitimate email) are more annoying than false negatives (letting spam through)
- **Criminal justice**: False positives (wrongly convicting innocent people) may carry different weight than false negatives (failing to convict guilty people)

**Accuracy for whom?** Different stakeholders bear different costs from different error types. A medical diagnostic test's false negatives have severe implications for patients, whereas false positives might burden the healthcare system with unnecessary costs.

## 67.2 Cost-Sensitive Learning

Rather than treating all errors equally, we can assign explicit costs to different error types.

| | $y = 0$ | $y = 1$ |
|---|---|---|
| $\hat{y} = 0$ | $c_{00}$ | $c_{01}$ |
| $\hat{y} = 1$ | $c_{10}$ | $c_{11}$ |

Figure 40: Confusion matrix with cost annotations. The cost matrix $c_{ij}$ weights different outcomes: $c_{00}$ and $c_{11}$ represent correct predictions (typically zero cost), while $c_{01}$ (false positive) and $c_{10}$ (false negative) represent errors with potentially different costs.

**Cost-Sensitive Loss**

Assign explicit costs to different error types:

$$\mathcal{L}_{\text{cost}} = \frac{1}{n}(FN \times c_{FN} + FP \times c_{FP}) = \frac{1}{n}(FN \times c_{10} + FP \times c_{01}) \tag{39}$$

The notation:

- $c_{01}$ = cost of Type I error (false positive)—predicted 1, actual 0
- $c_{10}$ = cost of Type II error (false negative)—predicted 0, actual 1

This forces you to **explicitly encode values**—what is the relative cost of a false positive versus a false negative? We want to achieve $c_{00}$ and $c_{11}$ (correct predictions), but we must decide how to balance $c_{10}$ and $c_{01}$ against each other.

This approach aims to minimise a weighted sum of errors, allowing for a more nuanced optimisation that reflects the actual costs (financial, ethical, etc.) associated with different errors. Libraries like scikit-learn implement this concept through mechanisms like "class weights."

## 67.3 Receiver Operating Characteristic (ROC) Curves

The ROC curve is a powerful tool for evaluating binary classification models. It provides a graphical representation of a classifier's ability to distinguish between classes at various threshold settings.

### 67.3.1 ROC Curve Construction

The ROC curve plots the True Positive Rate (TPR) against the False Positive Rate (FPR) at different threshold levels. Here, the risk score $\hat{r}(x)$ acts as the threshold—the point at which the predicted probability is considered sufficient to classify an observation into the positive class.

**ROC Curve Components**

At threshold $\tau$:

$$\text{True Positive Rate (TPR)} = \frac{TP}{TP + FN} \quad \text{(sensitivity, recall)} \tag{40}$$

$$\text{False Positive Rate (FPR)} = \frac{FP}{FP + TN} \quad \text{(1 - specificity)} \tag{41}$$

TPR measures how well we identify actual positives; FPR measures how often we incorrectly flag negatives as positive.

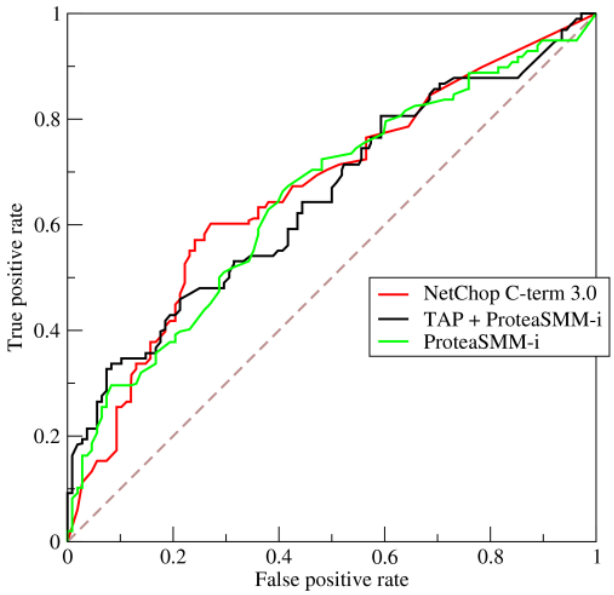



Figure 41: ROC curve interpretation. Bottom-left: nothing predicted positive (threshold = 1). Top-right: everything predicted positive (threshold = 0). Diagonal: random classifier (AUC = 0.5). Closer to the top-left corner indicates better performance.

### 67.3.2 Model Comparison with ROC Curves

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 "reasonable loss," that model is preferred.

**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. This approach aligns with treating classification as thresholded regression, where the goal is to accurately predict numerical probabilities rather than just discrete classes.

### 67.3.3 Area Under the Curve (AUC)

**Area Under Curve (AUC)**

- **AUC = 1**: Perfect classifier
- **AUC = 0.5**: Random classifier (no better than chance)
- **AUC provides threshold-independent evaluation**—an aggregate measure of performance across all possible thresholds

**Important caveat**: AUC is an overall evaluation of a model, but often we are interested in specific regions of the curve (e.g., 95%+ TPR). AUC tells you about performance over *all* thresholds, which may not match your specific operational requirements.

**Model Selection Strategies**

AUC allows us to select **both the model and the threshold**:

- The model gives you the ROC curve
- Each point on the curve corresponds to a threshold

**Threshold-led approach**:

1. Specify constraints (e.g., "cannot accept FPR $> 20\%$")
2. Restrict attention to the acceptable region of the x-axis
3. Choose the model with highest TPR in that constrained region

**Model-led approach**:

1. Compare AUC across models to select the best overall model
2. Then choose an appropriate threshold for your application

# 68 Discrimination in Classification

A fundamental problem in machine learning fairness is that features $X$ can encode sensitive information about group membership, either directly or indirectly.

## 68.1 How Discrimination Arises

**Explicit encoding**: Features directly encode sensitive attributes (race, gender, age). 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. Models can learn discriminatory patterns through features that are correlated with group membership, even when sensitive attributes are not directly included. For example, socioeconomic factors can predict race quite well despite not using race directly.

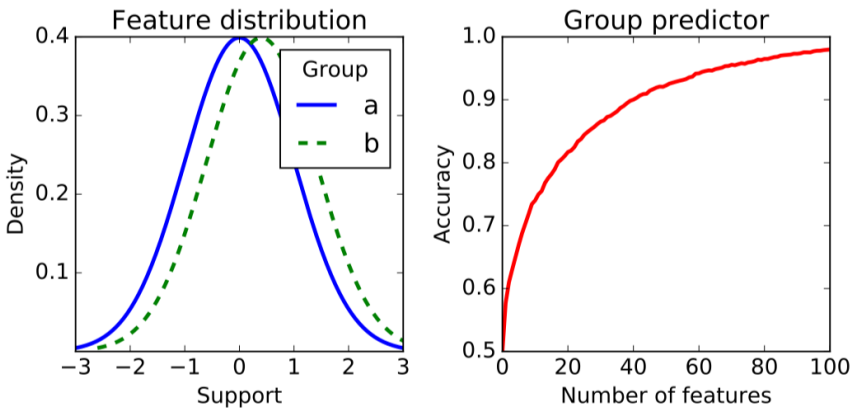



Figure 42: Accumulation of slight predictivity: Groups A and B may be similar on any single feature, making group membership hard to predict from one feature alone. But with many features, each slightly predictive of group membership, we can predict group membership extremely well.

### 68.2 Accumulation of Slight Predictivity

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 is related to **redundant encodings**—combinations of non-sensitive features can effectively encode sensitive information.

**NB!**

**Redundant encodings**: Even if you remove the sensitive attribute from your feature set, combinations of other features can reconstruct it. Simply dropping race or gender from your model doesn't guarantee fairness—the model may learn to discriminate based on proxies that are highly correlated with the protected attribute.

### 68.3 Approaches to Addressing Discrimination

Several strategies exist for addressing discrimination in classification:

- **Fairness metrics and objectives**: Various metrics (demographic parity, equal opportunity, etc.) can guide evaluation and model adjustment
- **Feature selection and engineering**: Carefully reviewing features to minimise encoding of sensitive information, directly or indirectly
- **Bias mitigation techniques**: Pre-processing (alter training data), in-processing (adjust learning algorithm), and post-processing (modify predictions) methods
- **Transparency and interpretability**: Understanding how models make decisions helps identify and address sources of bias

## 69 Quantitative Fairness Criteria

We now formalise three major approaches to quantitative fairness. Each imposes different restrictions on the relationship between the risk score, sensitive attributes, and outcomes.

**Notation for Fairness Criteria**

- $R$: Risk score—the model's output, e.g., $\hat{r}(x)$
- $A$: Sensitive attribute—e.g., race, gender (binary: $A \in \{0, 1\}$)
- $Y$: True outcome/label—e.g., good/bad job candidate
- $\hat{Y}$: Predicted label—e.g., the decision to hire
- $X$: Features of the prediction model

Our goal is to understand restrictions on $R$ which would lead to "fair" results.

## 69.1 Independence (Demographic Parity)

**Independence**

$$R \perp A \tag{42}$$

**The risk score is independent of the sensitive attribute.** This implies the probability distribution of risk scores is identical across groups:

$$P(R \mid A = 0) = P(R \mid A = 1) \tag{43}$$

**Implication**: Equal acceptance rates across groups. If we threshold $R$ to make decisions, the proportion of positive decisions will be the same for both groups.

Independence ensures that group membership doesn't affect the probability of receiving a positive prediction. Also called **demographic parity** or **statistical parity**.

### 69.1.1 Implications for Fairness

**Risk scores and group membership**: If the risk score is independent of the sensitive attribute, the model evaluates risk based solely on factors that do not include group membership. The decision-making process is "fair" in the sense that both groups receive positive predictions at equal rates.
**Acceptance rates across groups**: A direct implication is that acceptance rates—the proportion of individuals from each group predicted to be in the positive class (e.g., receiving a loan, being hired)—should be the same across groups.

**Achieving Independence: Orthogonal Projection**

One technique to enforce independence is to remove the influence of $A$ from predictors:

1. Regress each predictor on the sensitive attribute $A$
2. Use the residuals from these regressions as new predictors

This process doesn't remove $A$ itself but removes the portion of variation in each feature that corresponds to $A$. The residuals represent the original predictors with the influence of the sensitive attribute removed, so predictors become uncorrelated with $A$.
This **partialling out** or **orthogonal projection** method aims to mitigate the impact of the sensitive attribute on predictions, reducing bias related to that attribute.

## 69.2 Separation (Equalised Odds)

**Separation**

$$R \perp A \mid Y \tag{44}$$

**The risk score is independent of the sensitive attribute, within strata defined by the true outcome.** This implies:

$$P(R \mid A = 0, Y = y) = P(R \mid A = 1, Y = y) \quad \forall y \in \{0, 1\} \tag{45}$$

**Implication**: Equal error rates across groups:

$$\text{TPR}_{A=0} = \text{TPR}_{A=1} \tag{46}$$
$$\text{FPR}_{A=0} = \text{FPR}_{A=1} \tag{47}$$

Separation ensures that individuals with the same true outcome $Y$ are treated similarly regardless of group membership.

### 69.2.1 Equal Treatment Among Similarly Situated Individuals

The risk score for individuals within the same outcome category (e.g., actually good or actually bad candidates) should be independent of $A$. For example, men and women who are genuinely good candidates should receive similar risk scores, irrespective of gender.
**Important**: This does *not* imply that the proportions of good and bad candidates need to be the same between groups. Different base rates are allowed—what matters is equal treatment within outcome strata.

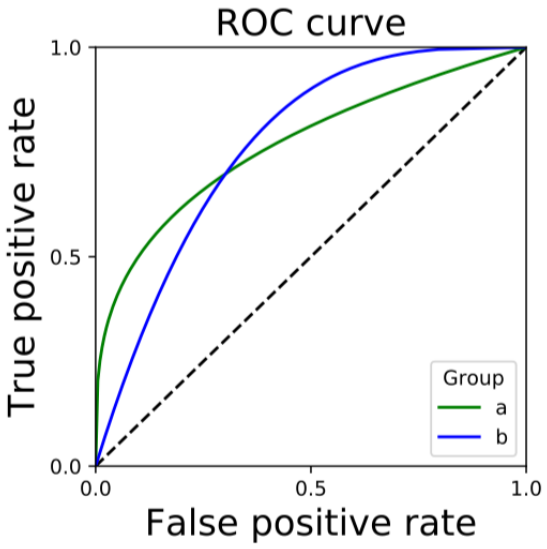



Figure 43: Separation requires operating at the same point on the ROC curve for both groups. Only the intersection of ROC curves for different groups satisfies separation—we must select a threshold that achieves equal TPR and FPR across groups.

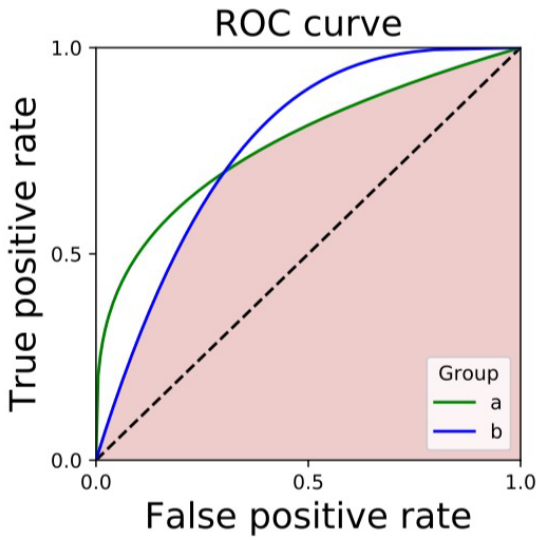



Figure 44: When groups have different base rates (different proportions of actual positives), achieving equal error rates typically requires using different thresholds for each group.

This approach to fairness—ensuring equal error rates across groups—aligns with the concept of **equalised odds**, a fairness criterion demanding that a classifier's TPR and FPR be equal across groups defined by a sensitive attribute.

### 69.3 Sufficiency (Calibration)

**Sufficiency**

$$Y \perp A \mid R \tag{48}$$

**Outcome frequency given risk score is equal across groups.** Given the same risk score, the probability of a positive outcome is equal regardless of group membership:

$$P(Y = 1 \mid R = r, A = 0) = P(Y = 1 \mid R = r, A = 1) \tag{49}$$

**Implication**: Each group is **well-calibrated**—predicted probabilities match actual outcome frequencies within each group.

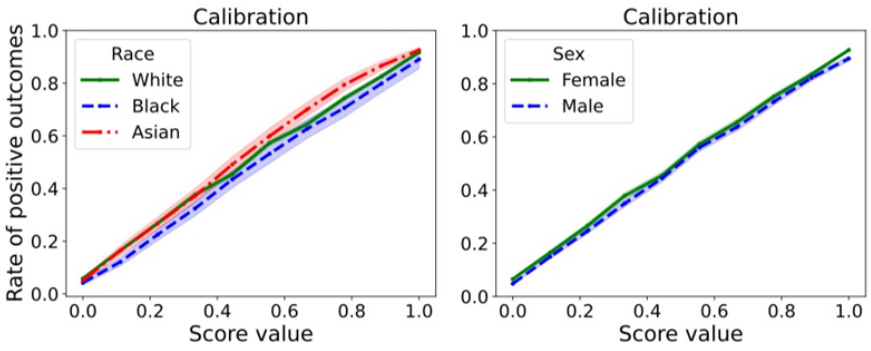



Figure 45: Calibration: A model predicting 70% probability should be correct approximately 70% of the time, and this should hold for each group separately. A well-calibrated model has predicted probabilities that match observed frequencies.

### 69.3.1 Understanding Calibration

**Calibration** measures how well a model's predicted probabilities correspond to actual outcomes. A well-calibrated model means that if the model predicts an event with 70% probability, that event should indeed happen approximately 70% of the time.
*On average*, the risk score is right in each group. However:

- This doesn't necessarily mean the model is perfect on an individual level
- It indicates that, on balance, risk scores are reliable indicators of true risk *within each group*
- A model can satisfy sufficiency but still have room for improvement in accuracy and precision for individual predictions

**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.

## 70 Impossibility Results

A central result in fairness research is that these three criteria cannot generally be satisfied simultaneously.

**NB!**

**Impossibility theorem**: If the sensitive attribute $A$ is related to the outcome $Y$ (i.e., base rates differ between groups), you **cannot simultaneously satisfy**:

- Independence AND Sufficiency
- Independence AND Separation

These fairness criteria are **mutually exclusive** when groups have different base rates. You must choose which notion of fairness matters most for your application.

## 70.1 Independence versus Sufficiency

**The Independence-Sufficiency Tradeoff**

**Independence** ($R \perp A$): Risk score distribution is the same across groups, implying equal selection/acceptance rates.
**Sufficiency** ($Y \perp A \mid R$): For any given risk score, outcome probabilities are equal across groups.
When base rates differ:

- **Good calibration ⇒ unequal acceptance rates**: If the model accurately reflects differing distributions of risk between groups, acceptance rates will naturally differ
- **Equal acceptance rates ⇒ poor calibration**: Forcing equal acceptance rates means the model no longer accurately reflects the true risk distributions

## 70.2 Independence versus Separation

**The Independence-Separation Tradeoff**

**Independence** ($R \perp A$): Equal acceptance rates across groups.
**Separation** ($R \perp A \mid Y$): Equal error rates (TPR and FPR) across groups.
When base rates differ:

- **Equal acceptance rates ⇒ unequal error rates**: The model disregards actual outcome distributions in favour of equalising decision rates, which can amplify disparities
- **Equal error rates ⇒ unequal acceptance rates**: The model adjusts predictions to compensate for differences in outcome distributions, leading to different acceptance rates

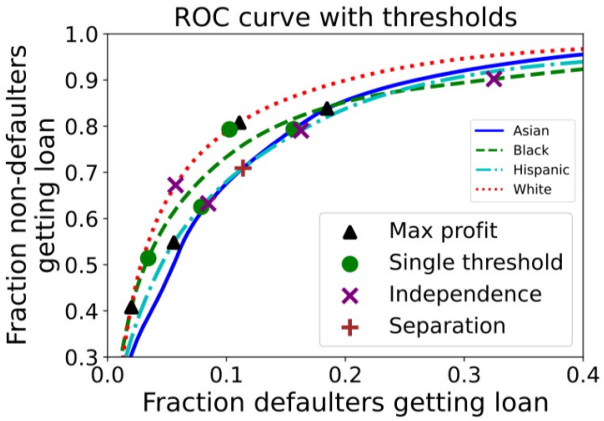



Figure 46: Different fairness criteria lead to different operating points on ROC curves and therefore different models. The choice of fairness criterion determines which model and threshold combination is "optimal."

# 71 Fairness is Not a Technical Problem

**Key Insight**

You cannot have it all. Reasonable quantitative measures of fairness conflict with one another. You must think through what matters in your particular case.
Fairness is a problem of expressing the **values** that your system should embody—you have to explicitly encode those values (as we saw with cost-sensitive learning).

## 71.1 Different Criteria, Different Models

Each fairness criterion leads to a different model:

- **Max profit**: No fairness constraints; pure accuracy optimisation. May deliver wildly different rates by group.
- **Single threshold**: One threshold applied uniformly to all groups
- **Independence**: Constrain to equal acceptance rates across groups
- **Separation**: Constrain to equal error rates across groups

The "right" choice depends on context, values, and stakeholder considerations—not on technical optimisation alone.

## 71.2 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"—this causes you to ignore its wider purpose and get distracted by technical details. Instead, evaluate the larger system the algorithm is part of:

- What outcomes does it produce in practice?
- Who benefits and who is harmed?
- What feedback loops does it create?
- What institutional incentives shape its use?

**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.

# 72 Summary

**Key Concepts from Week 7a: Quantitative Fairness**

1. **Risk scores**: Classification uses regression to estimate $P(Y = 1 \mid X)$, then thresholds to produce decisions
2. **Cost-sensitive learning**: Explicitly encode the relative costs of different error types rather than treating all errors equally
3. **ROC curves**: Visualise TPR/FPR tradeoff across thresholds; AUC provides threshold-independent evaluation
4. **Implicit encoding**: Removing sensitive features doesn't prevent discrimination—proxies and redundant encodings can reconstruct protected attributes
5. **Three fairness criteria**:
   - Independence (demographic parity): $R \perp A$ — equal acceptance rates
   - Separation (equalised odds): $R \perp A \mid Y$ — equal error rates
   - Sufficiency (calibration): $Y \perp A \mid R$ — calibrated predictions per group
6. **Impossibility**: These criteria conflict when base rates differ—you cannot satisfy all simultaneously
7. **Values matter**: Fairness requires explicit choices about which tradeoffs are acceptable, not just algorithmic optimisation
8. **POSIWID**: Evaluate systems by their actual outcomes, not stated intentions

# 73 Decision Trees

Decision trees are fundamental building blocks for constructing prediction functions in both classification and regression settings. The core idea is simple: partition the feature space into a number of disjoint regions, and make predictions based on the training observations that fall within each region.

**Decision Tree Prediction**

Partition the feature space into $J$ disjoint regions $R_1, \ldots, R_J$. Each region is defined by a sequence of threshold conditions on features, forming a hierarchical structure.
For a test point $\tilde{x}$:

$$f(\tilde{x}) = \sum_{j=1}^{J} \hat{y}(R_j) \cdot \mathbf{1}[\tilde{x} \in R_j]$$

where the prediction within each region is:

$$\hat{y}(R_j) = \frac{\sum_{i=1}^{n} y_i \, \mathbf{1}[x_i \in R_j]}{\sum_{i=1}^{n} \mathbf{1}[x_i \in R_j]}$$

This is simply the average outcome (regression) or majority class (classification) of training points in region $R_j$.

A decision tree can be visualised as a flowchart: starting at the root node, we traverse down the tree by answering questions at each internal node ("Is feature $x_j \leq t$?") until we reach a terminal node (leaf) that provides our prediction.

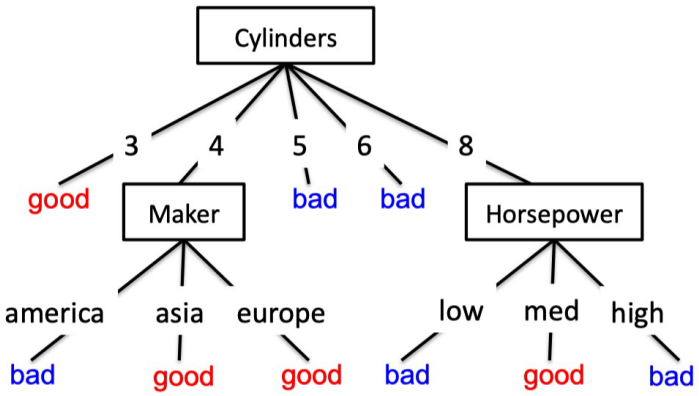



Figure 47: Decision tree structure. Each path from root to leaf defines a region through a sequence of threshold conditions. Note the interaction effects captured along branches—for example, "4 cylinders × continent" or "horsepower × low-med-high" combinations.

## 73.1 Constructing a Decision Tree

The tree-building process follows a recursive partitioning strategy:

1. **Choose a Feature**: At each node, the algorithm selects the feature that "best" splits the current set of observations. The criterion for "best" depends on the task:
   - **Regression**: Variance reduction (minimise MSE within resulting nodes)
   - **Classification**: Gini impurity or information gain (entropy reduction)

2. **Split the Data**: Partition observations based on the chosen feature's values. For continuous features, this typically involves a threshold split; for categorical features, branches may correspond to unique values.
3. **Recurse**: Apply the same procedure to each child node, creating a hierarchical structure of increasingly refined partitions.
4. **Make Predictions at Leaves**: When a stopping criterion is reached (maximum depth, minimum samples, or no further improvement), terminal nodes produce predictions:
   - **Classification**: The most common class label among training samples in that leaf
   - **Regression**: The average of target values for training samples in that leaf

**Key Intuition**

Decision trees learn a piecewise constant function. The prediction surface consists of axis-aligned rectangular regions, each with a constant predicted value. This explains why decision tree predictions appear "blocky" or "pixelated" when visualised.

## 73.2 Properties of Decision Trees

**Advantages**

- **Flexibility**: A very general hypothesis class that can:
  - Handle any input type (continuous, categorical, ordinal)
  - Capture complex nonlinear relationships without explicit specification
  - Model interaction effects naturally through branch combinations
- **Interpretability**: One of the most interpretable model classes:
  - Visualise as a flowchart understandable by non-experts
  - Explain predictions as a sequence of yes/no questions
  - Identify which features matter for specific predictions
- **Non-parametric**: No assumptions about functional form between features and response
- **No standardisation needed**: Splits are based on thresholds, so feature scaling is irrelevant
- **Fast and scalable**: Greedy construction is computationally efficient, making trees practical for large datasets
- **Robust to outliers**:
  1. An outlier only affects predictions within its own leaf node
  2. A single outlier typically won't change the internal structure of splits—the tree's architecture remains stable
- **Handle missingness naturally**: Can create splits based on "Is feature $x_j$ missing?" without imputation

**NB!**

**Disadvantages**:

- **Overfitting**: Without constraints, trees will grow until training error reaches zero, perfectly memorising the data including noise
- **High variance**: Small changes in training data can produce dramatically different tree structures
- **Greedy optimisation**: Finding the globally optimal tree is NP-hard; practical algorithms find locally optimal solutions that may be globally suboptimal
- **Suboptimal accuracy**: The simple hierarchical decision-making process may not capture all nuances in data—single trees often underperform other methods
- **Instability to data changes**: Adding, removing, or slightly modifying a few training points (especially near decision boundaries) can fundamentally alter the tree structure
- **Instability to feature changes**: Due to hierarchical dependence on features, small changes to the feature set (adding, removing, or modifying features) can cascade into very different trees

**Summary**: Decision trees are **high-variance learners**. This is their fundamental weakness—and the motivation for ensemble methods.

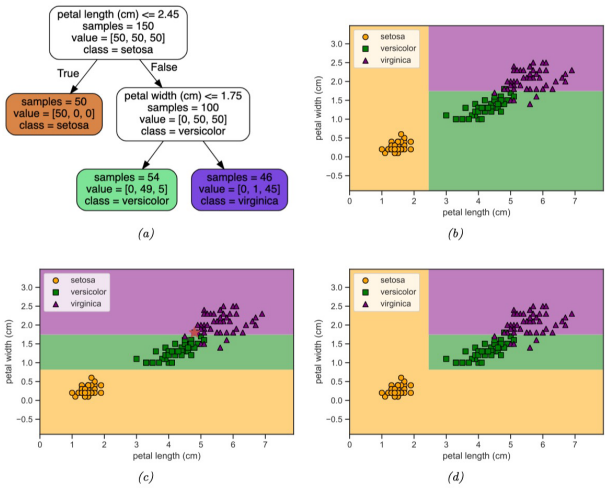



Figure 48: High variance illustrated: small changes in training data lead to very different tree structures. The same underlying relationship is modelled with dramatically different decision boundaries.

**To mitigate these weaknesses**: pruning, ensemble methods (bagging, random forests, boosting), and careful cross-validation for hyperparameter selection.

## 74 Splitting Strategies

The type of split depends on whether the feature is categorical or continuous.

## 74.1 Categorical Features: Splitting by Unique Value

For a categorical feature with $k$ unique values, we can partition the data into $k$ branches, one for each category.

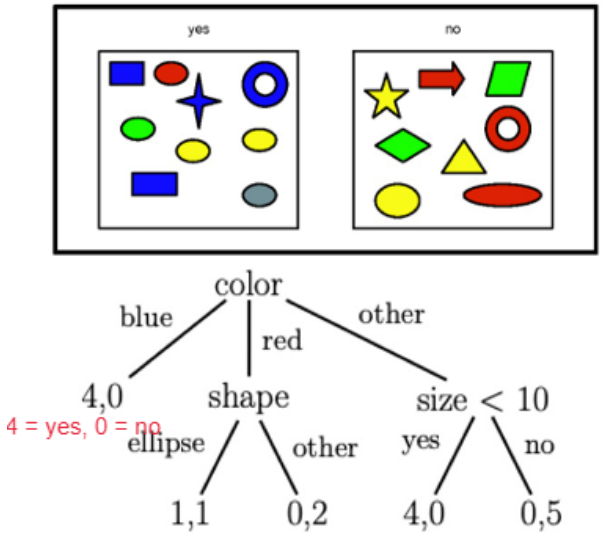



Figure 49: Categorical split: one branch per unique value. Each branch captures observations with that specific category.

**Advantages**: Straightforward; ensures the model captures the effect of each category on the outcome.
**Disadvantages**: Can lead to very complex trees, especially with high-cardinality features. Many branches may contain few observations, making predictions unreliable and the model prone to overfitting noise.

**NB!**

High-cardinality categorical features (e.g., postcodes, user IDs) can fragment the data excessively. Consider grouping rare categories or using binary splits (category $\in S$ vs. category $\notin S$) instead of multi-way splits.

## 74.2 Continuous Features: Splitting by Threshold

For continuous (or ordinal) features, we use binary threshold splits: observations with $x_j \leq t$ go left; observations with $x_j > t$ go right.

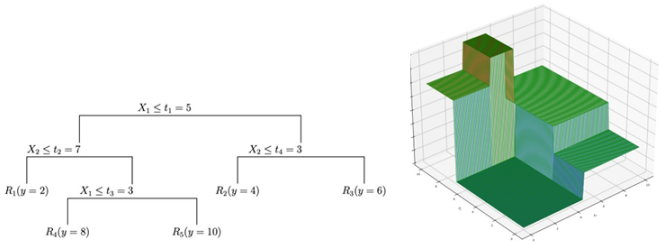



Figure 50: Threshold split for a continuous feature. The prediction function is flat within each terminal node and discontinuous at boundaries—this is the characteristic "stepped" shape of tree predictions.

**Advantages**:

- More scalable than multi-way splits
- Produces simpler, more interpretable trees
- Allows identification of critical threshold values

**Disadvantages**: Requires searching over all possible thresholds to find the optimal split point—computationally intensive for features with many unique values.

**Threshold Search**

For a continuous feature with $n$ unique sorted values $v_1 < v_2 < \cdots < v_n$, evaluate candidate thresholds $t \in \{(v_i + v_{i+1})/2 : i = 1, \ldots, n-1\}$. This gives $O(n)$ candidates per feature, and $O(np)$ candidates per node for $p$ features.

# 75 Optimising Trees

**NB!**

Finding the globally optimal decision tree is **NP-hard**. The space of possible trees grows exponentially with the number of features and thresholds. For any non-trivial dataset, exhaustively searching all possible trees is computationally infeasible.
This means we must use **heuristic** algorithms—primarily greedy approaches—that find good (but not necessarily optimal) solutions.

## 75.1 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 $\theta = \{(j_1, t_1), (j_2, t_2), \ldots\}$ 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. This rules out standard optimisation techniques like gradient descent.

**Approaches to Tree Optimisation**

Given non-differentiability, we have several options:

1. **Greedy recursive splitting**: Make the locally optimal choice at each node—by far the most common approach
2. **Dynamic programming**: For a fixed, small tree depth, one could theoretically enumerate all possible trees. This is computationally feasible only for very shallow trees
3. **Randomisation**: Techniques like random forests inject randomness (bootstrap sampling, feature subsampling) to explore a broader model space
4. **Pruning**: Grow a large tree greedily, then simplify post-hoc based on validation performance

## 75.2 Greedy Recursive Splitting

"If you can't be smart, be greedy." The standard approach is to make the best possible decision at each step, without considering future splits.

**Greedy Algorithm**

1. **Start at the root**: All training data begins in a single node
2. **Evaluate all possible splits**: For each feature $j$ and each candidate threshold $t$, compute a split quality metric
3. **Choose the best split**: Select the $(j^*, t^*)$ pair that optimises the metric
4. **Partition the data**: Create two child nodes containing observations with $x_{j^*} \leq t^*$ and $x_{j^*} > t^*$
5. **Recurse**: Apply steps 2–4 to each child node
6. **Stop**: When a stopping criterion is met (maximum depth, minimum samples per leaf, minimum samples to split, or no improvement possible)

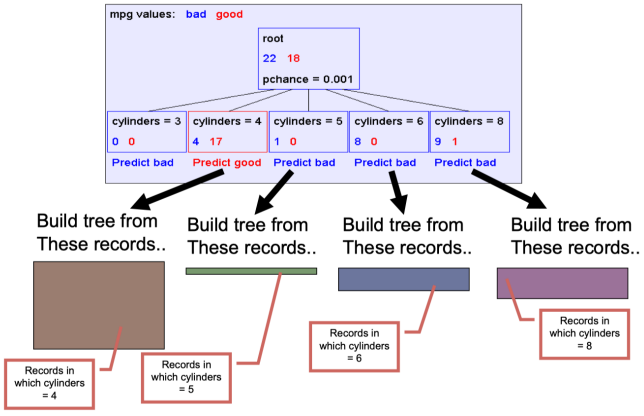



Figure 51: Greedy recursive splitting: at each node, we choose the locally optimal split without regard to how this affects future splits or the overall tree structure.

## 75.3 Evaluating Split Quality

How do we determine which split is "best"? For regression tasks, we use Mean Squared Error (MSE); for classification, Gini impurity or information gain.

**Split Quality for Regression: Weighted MSE**

Consider splitting dataset $\mathcal{D}$ into $\mathcal{D}_{\text{left}}$ and $\mathcal{D}_{\text{right}}$ based on condition $x_j \leq t$.
**Step 1**: Compute the MSE within each subset. For $\mathcal{D}_{\text{left}}$:

$$\text{MSE}_{\text{left}} = \frac{1}{|\mathcal{D}_{\text{left}}|} \sum_{i:x_i \in \mathcal{D}_{\text{left}}} (y_i - \bar{y}_{\text{left}})^2$$

where $\bar{y}_{\text{left}}$ is the mean of $y$ values in $\mathcal{D}_{\text{left}}$. Similarly for $\mathcal{D}_{\text{right}}$.
**Step 2**: Compute the weighted average MSE:

$$\text{MSE}_{\text{split}} = \frac{|\mathcal{D}_{\text{left}}|}{|\mathcal{D}|} \cdot \text{MSE}_{\text{left}} + \frac{|\mathcal{D}_{\text{right}}|}{|\mathcal{D}|} \cdot \text{MSE}_{\text{right}}$$

The weights account for the relative sizes of the two subsets.
**Step 3**: Choose the split $(j^*, t^*)$ that minimises $\text{MSE}_{\text{split}}$.
For **classification**, replace MSE with Gini impurity:

$$\text{Gini}(S) = 1 - \sum_{k=1}^{K} p_k^2$$

where $p_k$ is the proportion of class $k$ in set $S$. Lower Gini indicates purer nodes.

**Greedy Nature**

The algorithm chooses the split that minimises error *at this step*, without considering how this choice affects subsequent splits. This local optimisation is computationally efficient but does not guarantee finding the globally optimal tree structure.
Despite this limitation, greedy algorithms produce highly effective trees in practice, offering an excellent balance between computational cost and model quality.

### 75.4 Greed Eventually Overfits

If we keep splitting without constraint, training MSE will monotonically decrease. Eventually, with enough splits, we can achieve $\text{MSE}_{\text{train}} = 0$—each leaf contains a single training point.
This is **overfitting**: the model memorises training data, including noise, and generalises poorly to new observations. The model's complexity (number of leaves) approaches $n$, the number of training points.

## 76 Pruning

Pruning techniques limit tree complexity to improve generalisation. There are two main approaches: **pre-pruning** (early stopping) and **post-pruning** (grow then cut back).

### 76.1 Pre-Pruning: Early Stopping

Stop growing the tree before it overfits by setting constraints:

**Pre-Pruning Hyperparameters**

- **Maximum depth**: Limit how many levels the tree can have
- **Minimum samples per leaf**: Require each terminal node to contain at least $k$ observations
- **Minimum samples to split**: Only split a node if it contains at least $m$ observations
- **Minimum impurity decrease**: Only split if the improvement exceeds a threshold

Tune these via cross-validation.

## 76.2 Post-Pruning: Cost-Complexity Pruning

Also called **weakest link pruning**. The idea is to first grow a large tree that overfits, then prune back to improve generalisation.

**Cost-Complexity Pruning Algorithm**

1. **Grow the full tree**: Allow the tree to fit the training data perfectly (or nearly so)
2. **Define the cost-complexity criterion**: For a subtree $T$ with $|T|$ terminal nodes:

$$C_\alpha(T) = \sum_{m=1}^{|T|} \sum_{x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha |T|$$

   The first term measures fit (training error); the second penalises complexity.

3. **Prune iteratively**: Starting from the leaves, evaluate each internal node. If removing the subtree below a node (replacing it with a leaf) decreases $C_\alpha(T)$, perform the pruning. The "weakest link" at each step is the node whose removal causes the smallest increase in training error per leaf removed.
4. **Tune $\alpha$ via cross-validation**: Different values of $\alpha$ produce different levels of pruning. Select the $\alpha$ that minimises validation error.

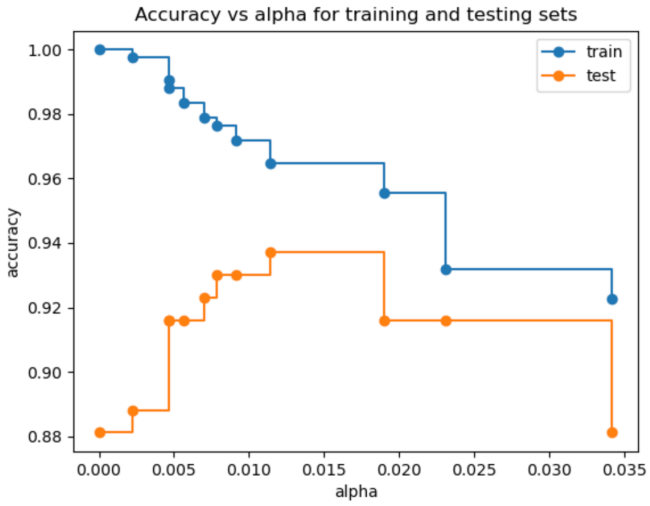



Figure 52: Cost-complexity pruning: the fully grown tree (left) has near-zero training error but high test error. As we prune (increase $\alpha$), test error initially improves before eventually increasing again when the tree becomes too simple.

**The Bias-Variance Trade-off in Pruning**

- **No pruning** ($\alpha = 0$): Low bias, high variance—tree overfits
- **Heavy pruning** (large $\alpha$): High bias, low variance—tree underfits
- **Optimal** $\alpha$: Balances bias and variance—found via cross-validation

# 77 Ensemble Methods

The high variance of decision trees motivates **ensemble methods**: combine many trees to reduce variance while maintaining (or even improving) predictive power.

**Ensemble Intuition**

Key insight: **Averaging many poor-but-diverse models can outperform a single good model**.
Each individual tree may have high variance (sensitive to the training data), but if their errors are uncorrelated, averaging their predictions smooths out the variance.

## 77.1 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 $n$ observations from the training data *with replacement*. Each bootstrap sample is the same size as the original but contains different observations (some repeated, some missing).
2. **Fit a decision tree to each sample**: Train a separate (typically unpruned) tree on each of the $M$ bootstrap datasets. Since each sample is different, each tree will be different.
3. **Aggregate predictions**:
   - **Regression**: Average the predictions: $\hat{f}(x) = \frac{1}{M}\sum_{m=1}^{M} f_m(x)$
   - **Classification**: Majority vote across trees

### 77.1.1 Bootstrap Sampling Properties

Each bootstrap sample contains, on average, approximately 63% of the unique observations from the original dataset. To see why:

**The 63% Rule**

The probability that observation $i$ is *not* selected in any of the $n$ draws is:

$$P(\text{not selected}) = \left(1 - \frac{1}{n}\right)^n \approx e^{-1} \approx 0.368$$

Therefore, each observation has approximately a 63% chance of appearing at least once in any given bootstrap sample.

### 77.1.2 Out-of-Bag (OOB) Estimates

The roughly 37% of observations not included in each bootstrap sample form the **out-of-bag** (OOB) set for that tree. This provides 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
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.

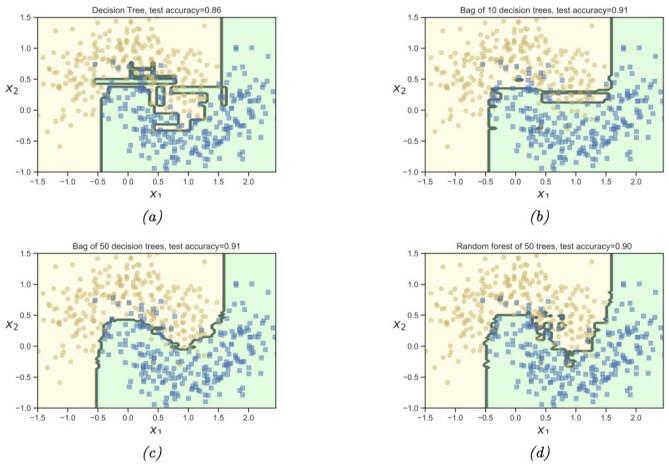



Figure 53: Bagging in action. (a) A single tree shows high variance with jagged, overfit predictions. As we aggregate more trees (b–d), the ensemble prediction smooths out, reducing variance while maintaining the ability to capture the underlying signal.

### 77.1.3 Why Bagging Works

**Variance Reduction through Averaging**

For $M$ independent random variables each with variance $\sigma^2$, their average has variance $\sigma^2/M$.
While bootstrap samples (and hence trees) are not fully independent, they are sufficiently different that averaging substantially reduces variance. Crucially:

- **Bias is preserved**: Each tree is (approximately) unbiased; the average is also unbiased
- **Variance is reduced**: Averaging reduces the high variance of individual trees
- **Parallelisable**: Trees are trained independently—easy to distribute across processors

### 77.1.4 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:

$$\widehat{\mathrm{Var}}[\hat{f}(x)] = \frac{1}{M-1}\sum_{m=1}^{M}\left(f_m(x) - \bar{f}(x)\right)^2$$

where $\bar{f}(x) = \frac{1}{M}\sum_m f_m(x)$ 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.

## 77.2 Random Forests

Random forests extend bagging with an additional source of randomness: **feature subsampling** at each split.

**Random Forest**

At each split in each tree:

1. Randomly select $m$ features from the full set of $p$ features
2. Find the best split considering *only* those $m$ features
3. Proceed with the split; repeat for child nodes with a fresh random subset

Typical choices for $m$:

- **Classification**: $m \approx \sqrt{p}$
- **Regression**: $m \approx p/3$

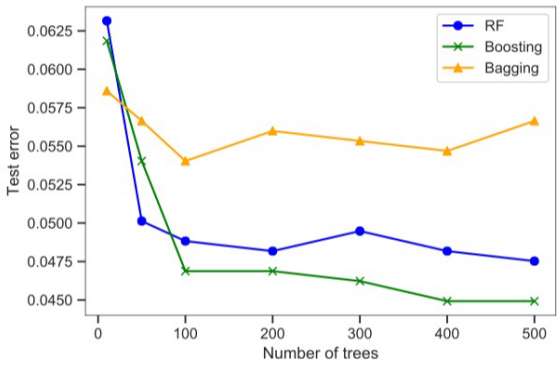



Figure 54: Random forest performance as trees are added. Feature subsampling decorrelates the trees, so adding more trees continues to improve performance. The ensemble captures complex structure that individual trees miss.

### 77.2.1 Why Feature Subsampling Helps

**The Decorrelation Effect**

Without feature subsampling, if one feature is strongly predictive, most trees will split on it first. This makes trees similar to each other—**correlated**.
Correlated trees make correlated errors. When we average correlated predictions, variance reduction is limited:

$$\text{Var}\left[\frac{1}{M}\sum_m X_m\right] = \frac{\sigma^2}{M} + \frac{M-1}{M}\rho\sigma^2$$

where $\rho$ is the correlation between trees. As $M \to \infty$, variance approaches $\rho\sigma^2$, not zero. **Feature subsampling reduces** $\rho$ by preventing the same features from dominating every tree. This allows greater variance reduction through averaging.

**Random Forest Trade-offs**

- **Individual trees are worse**: Not always considering all features increases bias per tree
- **But trees are less correlated**: When aggregated, the variance reduction is greater than with bagging alone
- **Net effect**: Usually superior to bagging, especially when a few features dominate
- **Scalability**: More trees generally help (diminishing returns, but no overfitting from adding trees)

Random forests are, in some sense, an **atheoretical heuristic**—there's no deep theoretical justification for why feature subsampling works so well. Yet empirically, random forests are remarkably effective across a wide range of problems, often achieving near-state-of-the-art performance with minimal tuning.

**NB!**

**Interpretability trade-off**: Random forests sacrifice the interpretability of single trees. You cannot easily visualise or explain why the model makes a particular prediction. For applications requiring explainability (e.g., medical diagnosis, credit decisions), this may be a significant limitation.
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.

### 77.3 Correlation Between Trees and the Path to Boosting

Even with feature subsampling, trees in a random forest can exhibit correlated predictions—particularly if certain features are overwhelmingly important or if the signal-to-noise ratio is low.
When trees make similar errors, averaging provides limited benefit. This observation motivates **boosting methods** (covered in the next section), which take a different approach: rather than training trees independently and averaging, boosting trains trees **sequentially**, with each new tree explicitly targeting the errors of the previous ensemble.

## 78 Summary

**Key Concepts from Week 7b: Tree-Based Methods**

1. **Decision trees** partition the feature space into regions and predict the mean (regression) or mode (classification) within each region
2. **Splitting strategies**: Categorical features use multi-way splits by unique value; continuous features use binary threshold splits
3. **Optimisation is NP-hard**: Trees are non-differentiable; we use greedy recursive splitting to find locally optimal solutions
4. **High variance**: Small data changes can produce dramatically different tree structures—the fundamental weakness of single trees
5. **Pruning**: Pre-pruning (early stopping) and post-pruning (cost-complexity) limit overfitting by controlling tree complexity
6. **Bagging**: Train trees on bootstrap samples; average predictions. Reduces variance while preserving (approximate) unbiasedness
7. **OOB error**: The $\sim$37% of observations not in each bootstrap sample provide free validation estimates
8. **Random forests**: Bagging + feature subsampling. Decorrelates trees, enabling greater variance reduction
9. **Variance estimation**: Disagreement across trees quantifies prediction uncertainty
10. **Next**: Boosting methods address correlated errors by sequential, error-correcting training (see Week 8)

# 79 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.

**Why Correlated Errors Are Problematic**

Recall the variance of a sum of random variables. If we have $M$ trees with predictions $F_1, \ldots, F_M$, each with variance $\sigma^2$, then the variance of their average is:

$$\text{Var}\left(\frac{1}{M}\sum_{m=1}^{M} F_m\right) = \frac{1}{M^2}\left(\sum_{m=1}^{M}\text{Var}(F_m) + 2\sum_{m<m'}\text{Cov}(F_m, F_{m'})\right)$$

If errors are uncorrelated ($\text{Cov}(F_m, F_{m'}) = 0$), this simplifies to $\sigma^2/M$—variance decreases as we add more trees.
If errors are positively correlated, the covariance terms are positive, and variance reduction is less effective. In the extreme case where all trees make identical errors, averaging provides no benefit whatsoever.

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. When trees independently fit the same data patterns, they tend to make similar mistakes. Boosting addresses this by **sequentially** fitting trees to correct previous errors—explicitly decorrelating the contribution of each new tree.

## 79.1 Ensemble Prediction Function

We can formalise the general ensemble method as a prediction function:

**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
- $\theta$ 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.

The key insight is that when we fit each $F_m(\cdot)$ independently, 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.

### 79.2 The Boosting Solution

**The Core Boosting Strategy**

1. Fit $F_1(\cdot)$ as normal
2. Fit $F_2(\cdot)$ on reweighted or residual-focused data that emphasises the errors made by $F_1(\cdot)$
3. Fit $F_3(\cdot)$ to correct errors made by $F_1 + F_2$
4. Continue: each subsequent tree focuses on learning different things than (i.e., correcting the errors of) the previous ensemble

This sequential, error-correcting approach reduces correlation between the errors of individual trees, leading to more robust and accurate ensemble methods.

How subsequent steps handle this error correction depends on the specific boosting method:

- **AdaBoost**: Increases the weights of instances that were misclassified by previous models, making them more "important" for the next model to get right
- **Gradient Boosting**: Each new model is trained on the residuals (differences between observed and predicted values) of the previous models, directly correcting ensemble errors

## 80 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 model
2. Calculate residuals (errors)
3. Fit a new model to the residuals
4. Add to ensemble; repeat

The residuals represent our mistakes. At each iteration, we fit a model that will correct these mistakes. Stack many **weak learners** to build a **strong learner**.

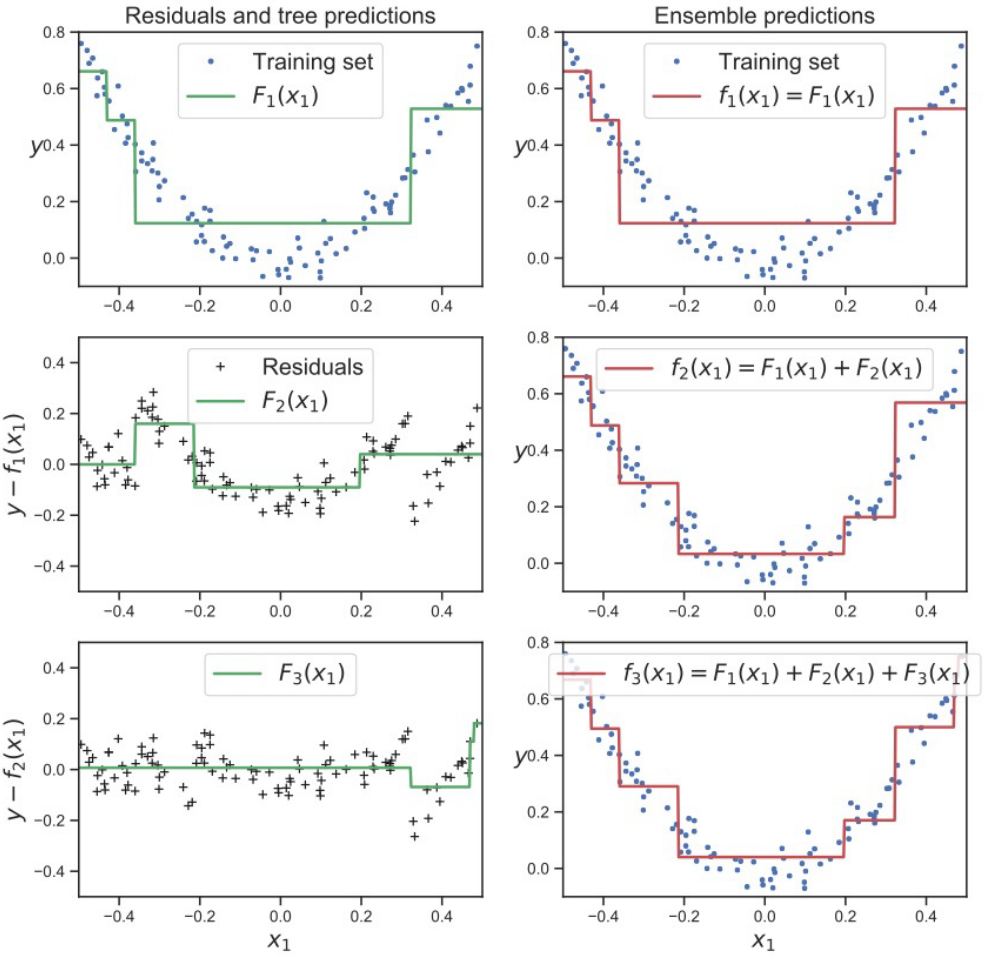



Figure 55: Green: single shallow decision tree. Red: combined trees. Each individual tree is not particularly expressive—often just depth 1–2—but when combined, they capture complex patterns. Cross-validation is used to select tree depth and number of iterations.

## 80.1 Weak Learners and Strong Learners

A key conceptual distinction in boosting:

- **Weak learner**: A model that performs only slightly better than random guessing. In boosting, each individual tree $F_m$ is a weak learner—shallow, simple, and not trying to fit the data perfectly.

- **Strong learner**: The final ensemble $f = \sum_m w_m F_m$ that combines all the weak learners. Despite each component being weak, their combination can be arbitrarily powerful.

We are not trying to fit a perfect model at each iteration. Rather, we reduce the error a bit each time. If we make $M$ large enough, we accumulate substantial error reduction: if each weak learner contributes something useful, combining them eventually produces a strong learner.

**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

## 80.2 Generic Boosting Loss Function

At iteration $m$, boosting solves the following optimisation problem:

**Generic Boosting Loss at Iteration $m$**

$$\beta_m, \theta_m = \arg\min_{\beta,\theta} \sum_{i=1}^{N} \mathcal{L}\left(y_i, \underbrace{f_{m-1}(x_i)}_{\text{previous ensemble}} + \underbrace{\beta}_{\text{learning rate}} \cdot \underbrace{F(x_i;\theta)}_{\text{new tree}}\right)$$

where:

- $f_{m-1}(x_i) = \sum_{j=1}^{m-1} \beta_j F_j(x_i)$ is the prediction from the ensemble at iteration $m-1$
- $F(x_i;\theta)$ is the new tree being fitted with parameters $\theta$
- $\beta$ 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)

The learning rate $\beta$ is crucial: it prevents overfitting by making small adjustments at each iteration rather than large corrections. 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 $\beta$ (how much each learner contributes)

## 80.3 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 from the previous iteration's prediction. Fit a new tree $F(x;\theta)$ to these residuals. This step focuses on learning from the mistakes of the ensemble thus far.
**Step 2: Finding the optimal $\beta$.** Once $F(x;\theta)$ is fitted to predict the residuals, find the optimal scaling factor $\beta$ 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 $\beta$ that best reduces the loss.
In practice, these steps are often simplified: $\beta$ may be treated as a fixed hyperparameter (set before training begins), and the tree fitting focuses purely on the residuals.

## 81 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.

**Least Squares Loss**

The loss function for least squares boosting is:

$$\ell(y, \hat{y}) = (y - \hat{y})^2$$

Inserting this into the generic boosting loss at iteration $m$:

$$\begin{aligned}\mathcal{L}\left(y_i, f_{m-1}(x_i) + \beta F(x_i;\theta)\right) &= \left(y_i - f_{m-1}(x_i) - \beta F(x_i;\theta)\right)^2 \\ &= \left(\underbrace{y_i - f_{m-1}(x_i)}_{\text{residual } r_i^{(m)}} - \beta F(x_i;\theta)\right)^2\end{aligned}$$

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)
2. For $m = 1, \ldots, M$:
   (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: $f_m = f_{m-1} + \beta F_m$

Note that the "previous model" $f_{m-1}$ is *all* of the previous trees combined—the ensemble prediction accumulates recursively.

## 82 AdaBoost

AdaBoost (Adaptive Boosting) is a boosting algorithm designed for binary classification that uses an exponential loss function.

**AdaBoost in One Sentence**

If the model misclassifies something, weight it up next time!

## 82.1 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.

**Label Encoding Transformation**

To transform binary labels from $\{0, 1\}$ to $\{-1, +1\}$:

$$\tilde{y} = 2y - 1$$

$$y = 0 \Rightarrow \tilde{y} = 2(0) - 1 = -1$$
$$y = 1 \Rightarrow \tilde{y} = 2(1) - 1 = +1$$

AdaBoost models produce predictions $F(x) \in (-\infty, +\infty)$. To interpret these as probabilities, we apply a logistic (sigmoid) function:

$$P(y = 1 \mid x) = \sigma(F(x)) = \frac{1}{1 + \exp(-2F(x))}$$

This maps real-valued predictions to the $(0, 1)$ interval.

## 82.2 The Exponential Loss Function

AdaBoost uses the exponential loss function:

**AdaBoost: Exponential Loss**

$$\ell(y, F(x)) = \exp(-yF(x))$$

where $y \in \{-1, +1\}$ and $F(x)$ is the ensemble prediction.
This heavily penalises confident wrong predictions:

- If $y = +1$ and $F(x) > 0$: loss $= \exp(-\text{positive}) < 1$ (small loss)
- If $y = +1$ and $F(x) < 0$: loss $= \exp(+\text{positive}) > 1$ (large loss)
- If $y = -1$ and $F(x) < 0$: loss $= \exp(-\text{positive}) < 1$ (small loss)
- If $y = -1$ and $F(x) > 0$: loss $= \exp(+\text{positive}) > 1$ (large loss)

The loss explodes exponentially when the model is confident and wrong.

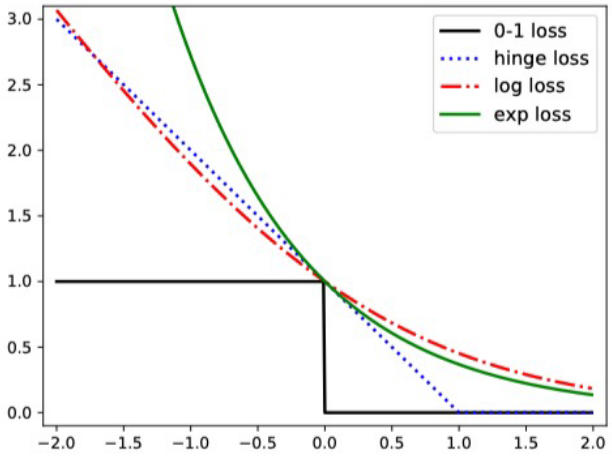



Figure 56: Comparison of loss functions for classification. The exponential loss (used by AdaBoost) penalises errors more aggressively than log loss. Both are differentiable surrogates for the 0-1 loss.

## 82.3 Comparing Loss Functions

**Log Loss vs Exponential Loss**

**Log-Loss (Logistic 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 bounded.

**Exponential Loss:**

$$\ell_{\exp}(y, F(x)) = \exp(-yF(x))$$

Used by AdaBoost. Penalty grows without bound as predictions move away from actual labels.

**Key difference**: Exponential loss imposes a higher penalty on incorrect predictions. The penalty increases exponentially with the degree of incorrectness, driving the model to prioritise correcting misclassifications.

Both serve as effective surrogate loss functions; minimising either over a sufficiently large dataset tends to lead to similar performance outcomes. However, from an optimisation perspective, log loss is generally easier to minimise due to its smoother gradient.

## 82.4 The AdaBoost Algorithm

**AdaBoost Algorithm (Discrete)**

At iteration $m$, AdaBoost minimises the exponential loss:

$$L_m = \sum_{i=1}^{N} \exp\left(-y_i \left(f_{m-1}(x_i) + \beta F(x_i)\right)\right)$$

This can be rewritten to highlight the sample weights:

$$L_m = \sum_{i=1}^{N} \omega_i^{(m)} \exp\left(-\beta y_i F(x_i)\right)$$

where the weight for sample $i$ at iteration $m$ is:

$$\omega_i^{(m)} = \exp\left(-y_i f_{m-1}(x_i)\right)$$

This weight represents the loss due to the previous ensemble's prediction—samples that were harder to classify in previous iterations become more significant in the current iteration.

The weights are a direct function of the previous loss, so we only need to optimise over the new tree $F_m$. This is what makes exponential loss particularly convenient for boosting.

**AdaBoost Update Rules**

**Step 1: Choose** $F_m$ to minimise weighted misclassification:

$$F_m = \arg\min_F \sum_{i=1}^{N} \omega_i^{(m)} \mathbf{1}[y_i \neq F(x_i)]$$

**Step 2: Compute weighted error rate**:

$$\text{err}_m = \frac{\sum_{i=1}^{N} \omega_i^{(m)} \mathbf{1}[y_i \neq F_m(x_i)]}{\sum_{i=1}^{N} \omega_i^{(m)}}$$

**Step 3: Compute the coefficient** $\beta_m$:

$$\beta_m = \frac{1}{2} \log\left(\frac{1 - \text{err}_m}{\text{err}_m}\right)$$

This formula ensures that more accurate models (lower $\text{err}_m$) have greater influence on the ensemble.
**Step 4: Update ensemble**: $f_m = f_{m-1} + \beta_m F_m$
**Step 5: Update sample weights** for the next iteration based on the new ensemble's performance.

**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

Where we had large loss before, we weight that up by the size of that loss. The exponential loss naturally implements this reweighting.

**NB!**

AdaBoost's aggressive penalisation of misclassified points makes it sensitive to outliers and mislabelled data. A single noisy observation that is consistently misclassified can accumulate enormous weight, potentially dominating the learning process.

# 83 Gradient Boosting

Gradient boosting generalises the boosting framework to work with any differentiable loss function, not just squared error or exponential loss.

**Gradient Boosting: The General Principle**

Rather than fitting residuals directly (as in least squares boosting) or reweighting samples (as in AdaBoost), gradient boosting fits each new tree to the **negative gradient** of the loss function.
This is gradient descent—but in function space rather than parameter space.

## 83.1 Relationship to Other Methods

It is helpful to understand how the boosting methods we have discussed relate to each other:

- **Least Squares Boosting**: A specific instance of gradient boosting using squared error loss. Particularly suited to regression problems.
- **AdaBoost**: Another specific instance, using exponential loss. Designed for classification with focus on reweighting misclassified instances.
- **Gradient Boosting**: The general framework. Its use of gradient descent on the loss function provides a systematic and generalisable approach to minimising prediction error.

## 83.2 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:

$$g_i^{(m)} = \left.\frac{\partial \ell(y_i, f(x_i))}{\partial f(x_i)}\right|_{f=f_{m-1}}$$

**Negative gradient**: The target for the new tree is:

$$\tilde{r}_i^{(m)} = -g_i^{(m)} = -\nabla_{f_{m-1}} \ell(y_i, f_{m-1}(x_i))$$

**Update rule**:

$$f_m = f_{m-1} + \beta_m F_m$$

where $F_m$ is fitted to approximate $\tilde{r}_i^{(m)}$.

**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 | Function space |
| Update | $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$ | $f \leftarrow f + \beta F$ |
| Gradient | Computed analytically | Approximated by tree $F$ |
| Step size | Learning rate $\eta$ | Shrinkage $\beta$ |

In gradient boosting, we cannot directly add the gradient to our function (we do not have explicit function representation). Instead, we fit a tree $F$ to approximate the negative gradient, then add that tree to our ensemble.

## 83.3 Why Fit the Negative Gradient?

For squared error loss:

$$\ell(y, f) = \frac{1}{2}(y - f)^2 \implies -\frac{\partial \ell}{\partial f} = y - f = \text{residual}$$

So for squared error, the negative gradient *is* the residual. Gradient boosting generalises this: for any loss function, the negative gradient tells us the direction to move our predictions to reduce loss.

**Practical Implementation of Gradient Boosting**

1. Initialise $f_0(x)$ (often to a constant, e.g., $\log(\bar{y}/(1-\bar{y}))$ for classification)
2. For $m = 1, \ldots, M$:
   (a) Compute negative gradients (pseudo-residuals):
   $$\tilde{r}_i^{(m)} = -\left.\frac{\partial \ell(y_i, f(x_i))}{\partial f(x_i)}\right|_{f=f_{m-1}}$$
   (b) Fit a regression tree $F_m$ to the pseudo-residuals $\{\tilde{r}_i^{(m)}\}$
   (c) (Optionally) perform line search to find optimal $\beta_m$
   (d) Update: $f_m(x) = f_{m-1}(x) + \beta_m F_m(x)$

For regression with squared loss, pseudo-residuals are simply residuals. For classification with log loss, they are related to the gradient of the logistic loss.

### 83.4 Hyperparameter Tuning

Gradient boosting has several important hyperparameters:

- **Learning rate** $\beta$: Controls the step size. Smaller values require more trees but often generalise better.
- **Number of trees** $M$: More trees reduce training error but risk overfitting.
- **Tree depth**: Deeper trees capture more complex interactions but may overfit.
- **Subsampling rate**: Fitting each tree on a random subsample adds regularisation.

These parameters interact: a smaller learning rate typically requires more trees. Cross-validation is essential for tuning.

## 84 XGBoost: Extreme Gradient Boosting

XGBoost extends gradient boosting with additional regularisation and computational optimisations. It has become one of the most successful machine learning algorithms in practice, particularly for structured/tabular data.

**XGBoost: Key Innovations**

XGBoost adds a suite of hyperparameters for regularisation and enables efficient cross-validation. The name "extreme" refers to pushing the limits of gradient boosting performance.

**XGBoost Enhancements**

**1. Explicit Regularisation**: Unlike traditional gradient boosting, XGBoost incorporates regularisation directly into the objective function:

$$\mathcal{L} = \sum_i \ell(y_i, f(x_i)) + \sum_m \Omega(F_m)$$

where the regularisation term is:

$$\Omega(F) = \gamma J + \frac{1}{2}\lambda \sum_{j=1}^{J} w_j^2$$

Here:

- $J$ = number of leaves in tree $F$
- $w_j$ = weight (prediction value) at leaf $j$
- $\gamma$ = penalty per leaf (encourages simpler trees)
- $\lambda$ = L2 regularisation on leaf weights

This provides more continuous control over tree complexity than simply limiting depth.
**2. Second-Order Approximation**: XGBoost uses both the gradient (first derivative) and the Hessian (second derivative) of the loss function:

$$\mathcal{L}^{(m)} \approx \sum_i \left[ g_i F_m(x_i) + \frac{1}{2} h_i F_m(x_i)^2 \right] + \Omega(F_m)$$

where $g_i = \partial \ell / \partial f$ and $h_i = \partial^2 \ell / \partial f^2$. This allows for:

- More accurate estimation of optimal step size and direction
- Better understanding of the loss landscape curvature
- More optimal split decisions

**3. Feature Subsampling**: Like Random Forests, XGBoost can sample features at each node before determining the best split. This:

- Contributes to diversity of models in the ensemble
- Reduces overfitting
- Speeds up computation

**4. Row Subsampling**: Training each tree on a random subsample of rows provides additional regularisation (similar to stochastic gradient descent).

**XGBoost vs Random Forests**

| | Random Forests | XGBoost |
|---|---|---|
| Base strategy | Bagging | Boosting |
| Tree fitting | Independent, parallel | Sequential, corrective |
| Tree depth | Typically deep | Typically shallow |
| Variance reduction | Averaging | Regularisation |
| Bias reduction | Limited | Strong (sequential correction) |
| Feature sampling | At each node | At each node (optional) |
| Interpretability | Moderate | Lower |

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.

# 85 Model Interpretation

Tree-based ensemble methods, while powerful, can be difficult to interpret. Two key techniques help us understand what these models have learned.

## 85.1 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, average over all trees and normalise:

$$R_k = \frac{1}{M} \sum_{m=1}^{M} R_k(T_m)$$

Normalise so that $\sum_k R_k = 100$ (or 1) for interpretability.

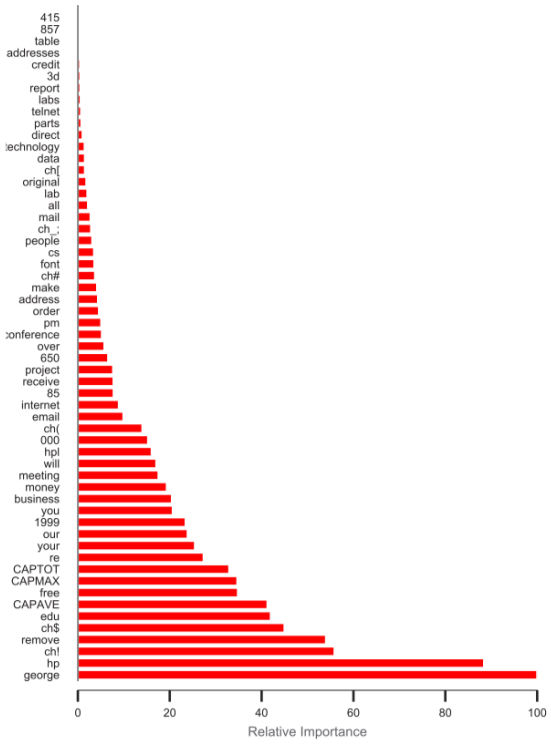



Figure 57: Feature importance for spam classification, where features are word occurrences. "George" is highly predictive—when the model splits on this feature, accuracy improves substantially.

**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.
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.

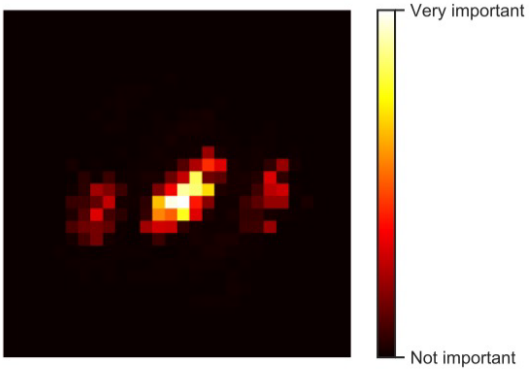



Figure 58: Feature importance for digit classification (3 vs 8), where features are pixel values. Intuitively, the middle pixels are most useful for distinguishing these digits.

## 85.2 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 prediction changes as one feature varies, averaging over all other features:

$$\bar{f}(x_k = c) = \frac{1}{N} \sum_{i=1}^{N} f(x_i \mid x_k^{(i)} = c)$$

**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
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.

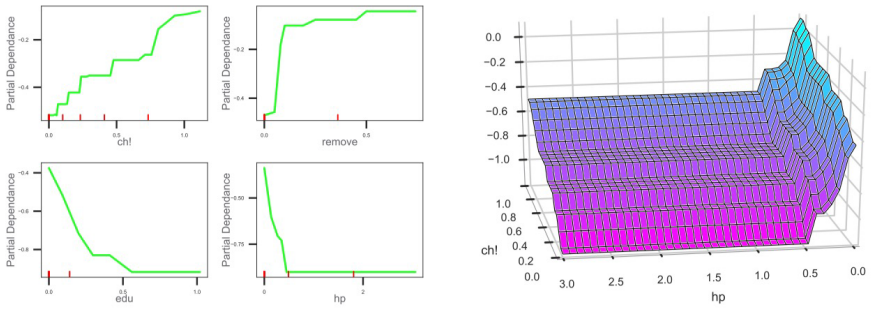



Figure 59: Partial dependence plots showing how predictions change as individual features vary. Each curve shows the average prediction when that feature is set to a specific value, averaging over all other features.

**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. Consider using Individual Conditional Expectation (ICE) plots or SHAP values for more nuanced interpretation.

## 86 Summary

**Key Concepts from Week 8: Boosting**

1. **Boosting**: Sequential ensemble method where each tree corrects errors of previous trees, explicitly decorrelating contributions
2. **Weak learners**: Simple models (shallow trees) that combine to form a strong learner
3. **Least squares boosting**: Fit trees to residuals; special case for squared error loss
4. **AdaBoost**: Uses exponential loss; reweights misclassified points to focus on hard cases
5. **Gradient boosting**: General framework; fit trees to negative gradient of any differentiable loss
6. **XGBoost**: Regularised gradient boosting with second-order approximations and feature subsampling
7. **Feature importance**: Which features contribute to accuracy (but not direction)
8. **Partial dependence**: How predictions change with one feature (reveals direction and shape)

**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.

**NB!**

**When boosting can fail**:

- Very noisy data with many outliers (especially AdaBoost)
- When the true relationship is best captured by linear models
- When interpretability is paramount (consider simpler models or careful use of explanation tools)
- When data has strong temporal dependencies (consider specialised time series models)

# 87 Overview

How should we choose which data to collect? This lecture examines sampling strategies 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**: Augmented Inverse Propensity Weighting (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; $y$ (efficacy) requires costly lab experiments

**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 $X$?

## 87.1 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 lecture

# 88 Data Leakage

**NB!**

**Data leakage**: Using information at training time that will not be available at prediction time.
This 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.
**Common Examples**:

- **Using future data to predict the past**: In time series, training on data from after the prediction point
- **Splitting correlated observations across train/test**: Same patient, same household, or same document appearing in both sets
- **Features that encode the outcome indirectly**: Variables that are only measured after the outcome is known
- **Preprocessing on full data**: Standardising or imputing using statistics computed on the entire dataset (including test set)

**Prevention**: Understand the production task you are solving. The train/test split should mirror exactly how the model will be deployed. Ask: "Are the features I am using actually available at the time I need to make a prediction?"

# 89 Random vs Non-Random Sampling

## 89.1 Why Random Sampling?

**Population Risk**

We care about **population risk**—the expected loss over the entire data distribution:

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

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 that Empirical Risk Minimisation (ERM) provides an unbiased estimate of population risk.

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 $R(f)$.

## 89.2 The Challenge of Heteroskedastic Noise

But what if some parts of the feature space are harder to learn than others?

**Heteroskedastic Noise**

Consider a model where the noise variance depends on $X$:

$$y_i = X_i\beta + \epsilon_i, \quad \epsilon_i \sim \mathcal{N}(0, \sigma^2(X_i))$$

The variance of the error term $\epsilon_i$ varies across observations. This heteroskedasticity makes some regions of $X$ inherently harder to predict:

- Areas with high noise ($\sigma^2(X_i)$ large) require more samples to achieve the same prediction accuracy
- Areas with low noise are easier to learn with fewer samples

This motivates **non-uniform sampling**—perhaps we should sample more heavily from high-noise regions.

## 89.3 Arguments for Random Sampling

Despite heteroskedasticity, random sampling has important benefits:

- **Bias reduction**: All data points have equal probability of inclusion, reflecting the true population distribution
- **Variance understanding**: The model is exposed to both high-noise and low-noise regions, learning to distinguish predictable from unpredictable areas
- **Model robustness**: Exposure to the full diversity of the data distribution makes the model more reliable

## 89.4 The Case for Non-Uniform (Adaptive) Sampling

The density of data points in different areas of feature space affects model accuracy. This suggests that uniform sampling may not always be optimal:

- **Improved accuracy in dense regions**: Where data is abundant, models tend to be more accurate because more data provides clearer signal
- **Non-uniform information distribution**: Not all regions contribute equally to model performance. Some areas may be critical for understanding complex phenomena or capturing rare but important events
- **Heteroskedastic considerations**: In high-noise regions, more data is needed to achieve comparable accuracy to low-noise regions

**Adaptive Sampling Strategies**

Several techniques address non-uniform information distribution:
**Stratified Sampling**: Divide the feature space into strata based on characteristics (e.g., density, noise level) and sample more from underrepresented or critical strata.
**Importance Sampling**: Weight data points based on their contribution to model learning, focusing training on more challenging or informative regions.
**Active Learning**: Dynamically select data points for labelling based on the model's current performance and uncertainties, focusing on areas that would most improve the model.

**NB!**

**Critical Distinction**:
Non-random sampling for **training data** can be corrected via reweighting (see Section 91). The bias introduced by non-uniform sampling can be mathematically removed.
Non-random sampling for **test data** is dangerous—you may have no idea how the model performs on the true population. There is no way to correct for not knowing what you do not know.

# 90 Active Learning

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.

**Active Learning Process**

1. **Initial model training**: Train a preliminary model on a small labelled set $\{(X_i, y_i)\}_{i=1}^{n_0}$
2. **Identify informative points**: Use the current model to score unlabelled points by their potential informativeness
3. **Request labels**: Obtain labels for the highest-scoring unlabelled points (typically from human annotators)
4. **Model update**: Retrain or update the model with the newly labelled data
5. **Repeat**: Continue until the labelling budget is exhausted or performance targets are met

The key question is: **how do we identify the most informative points?**

## 90.1 Criteria for Selecting Data Points

The goal is to reduce population risk by selecting points that will most improve the model. Several strategies exist:

- **Uncertainty sampling**: Choose points where the model is most uncertain (detailed below)
- **Query by committee**: Maintain multiple models and choose points where they disagree most—high disagreement suggests informative regions
- **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 of the data distribution

## 90.2 Uncertainty Sampling

The simplest and most common active learning approach: select points where the model is most uncertain about its prediction.

**Uncertainty Measures (Multi-Class Classification)**

Given predicted class probabilities $p_1, p_2, \ldots, p_C$, with $p^* = \max_c p_c$ (probability of most likely class):

**Maximum entropy**: $H = -\sum_c p_c \log p_c$

- High when predictions are spread across many classes
- Considers the full probability distribution
- Best when any misclassification is equally costly

**Margin sampling**: $p^* - p_{\text{second}}$

- Low when the model cannot distinguish between the top two classes
- Useful for refining decision boundaries between classes

**Least confident**: $1 - p^*$

- 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

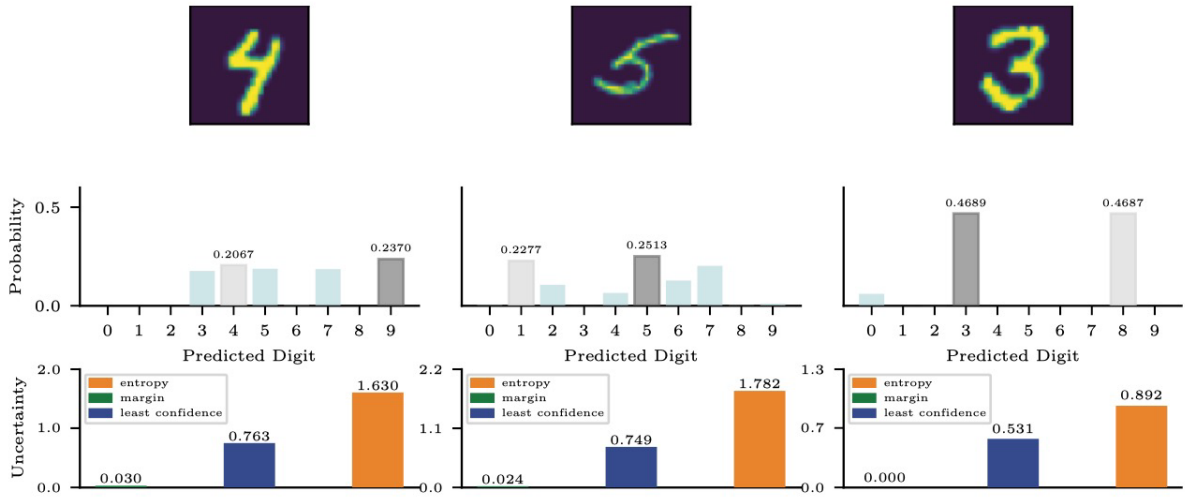



Figure 60: Uncertainty sampling in action: points near the decision boundary (where the model is uncertain) are selected for labelling. These are precisely the points where additional labels provide the most information for refining the boundary.

## 90.3 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**

BALD selects points that maximise the expected information gain about model parameters:

$$\mathbb{E}_{y|x,\mathcal{D}}\left[D_{KL}\left[p(\theta|\mathcal{D},x,y)\,\|\,p(\theta|\mathcal{D})\right]\right]$$

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.

**BALD: Uncertainty Decomposition**

Equivalently, BALD decomposes total predictive uncertainty:

$$\underbrace{H(y|x,\mathcal{D})}_{\text{Total uncertainty}} - \underbrace{\mathbb{E}_{p(\theta|\mathcal{D})}[H(y|x,\theta)]}_{\text{Aleatoric (irreducible)}}$$

- **First term**: Total uncertainty in predicting $y$ given $x$ (what uncertainty sampling uses)
- **Second term**: Average uncertainty when we know the true parameters—this is the inherent noise we cannot reduce
- **Difference**: Epistemic uncertainty—the part we *can* reduce by collecting more data

BALD targets points with high epistemic uncertainty, not just high total uncertainty.

## 91 Correcting for Non-Uniform Sampling

When we sample non-uniformly for training data, we need to correct the resulting bias to recover unbiased estimates of population risk.

### 91.1 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.

### 91.2 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:

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

**Intuition**:

- Points sampled with **low probability** $p_i$ were "lucky" to be included—they represent many similar unsampled points, so receive **high weight** $1/p_i$
- Points sampled with **high probability** $p_i$ are over-represented, so receive **low weight** $1/p_i$

This reweighting "undoes" the sampling bias, making the weighted sample behave like a random sample.

**Why does this work?** 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**

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)

The trade-off: IPW can have high variance when some $p_i$ values are very small (leading to very large weights).

## 92 Leverage Score Sampling

For ordinary least squares (OLS) regression, some observations are inherently more influential than others. Leverage score sampling exploits this structure.

**Leverage Scores**

The leverage of observation $i$ is defined as:

$$h_{ii} = x_i^\top (X^\top X)^{-1} x_i$$

This is the $i$th diagonal element of the **hat matrix** $H = X(X^\top X)^{-1}X^\top$.
**Interpretation**: Leverage measures how much changing $y_i$ would affect the fitted value $\hat{y}_i$:

$$\frac{\partial \hat{y}_i}{\partial y_i} = h_{ii}$$

High-leverage points have outsized influence on the regression fit—they are the observations that "pull" the regression line most strongly.

**Properties of Leverage Scores**

- Always between 0 and 1: $0 \leq h_{ii} \leq 1$
- Sum to the number of parameters: $\sum_i h_{ii} = p$
- High leverage often indicates points far from the centre of the $X$ distribution (outliers in feature space)
- High-leverage points provide the most "bang for buck" in determining the regression fit

## 92.1 Leverage Score Sampling Strategy

The insight: if we can only label a subset of observations, we should prioritise high-leverage points.

**Leverage Score Sampling Theorem**

Sample points with probability proportional to their leverage scores.
**Result** (Cohen & Peng, 2014): With $O(p \log n/\epsilon^2)$ samples selected by leverage, predictions achieve accuracy within $(1+\epsilon)$ of full-data accuracy.
This is remarkable: the required sample size depends on $p$ (number of features) but **not on** $n$ (population size). By focusing on influential points, we can achieve near-full-data accuracy with a sample size that is essentially independent of the population.

**When to Use Leverage Score Sampling**

Leverage score sampling is most useful when:

- Labels are expensive and we need efficient predictions
- The goal is prediction accuracy rather than understanding the full population distribution
- We have access to all features $X$ upfront but need to choose which $y$ values to measure

# 93 Random Fourier Features

Kernel methods are powerful but computationally expensive: solving kernel regression requires $O(n^3)$ time (for matrix inversion of the $n \times n$ Gram matrix). Random Fourier Features provide an efficient approximation.

## 93.1 The Computational Challenge

**The Gram Matrix Problem**

Kernel methods transform input data into a higher-dimensional space where linear methods become more powerful. However, this 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. Leverage score sampling can help by selecting a representative subset, but Random Fourier Features offer an alternative approach.

## 93.2 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)
- $\sigma$: Kernel bandwidth parameter

**The key insight**: This approximation works because shift-invariant kernels can be written as the Fourier transform of a probability distribution (Bochner's theorem). By sampling from this distribution, we create explicit random features that approximate the kernel.

**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 $\phi(x) \in \mathbb{R}^R$, then run standard linear regression on these transformed features. This reduces complexity dramatically while maintaining good approximation quality.

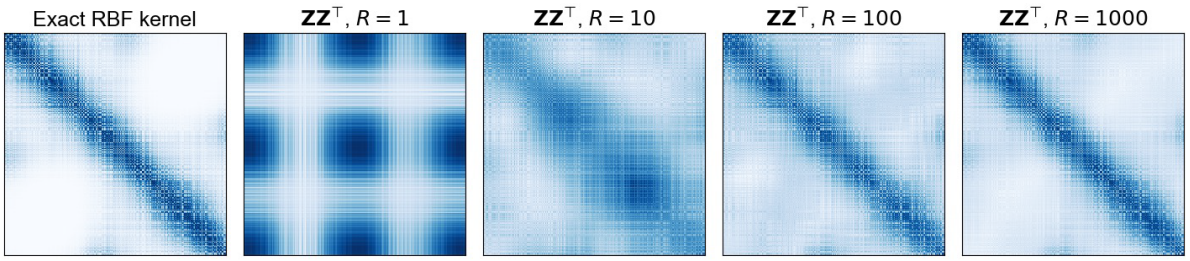



Figure 61: RFF approximation quality improves with more random features. Panel 1 shows the true RBF kernel function we aim to approximate. Subsequent panels show how the approximation improves as we increase the number of random features $R$.

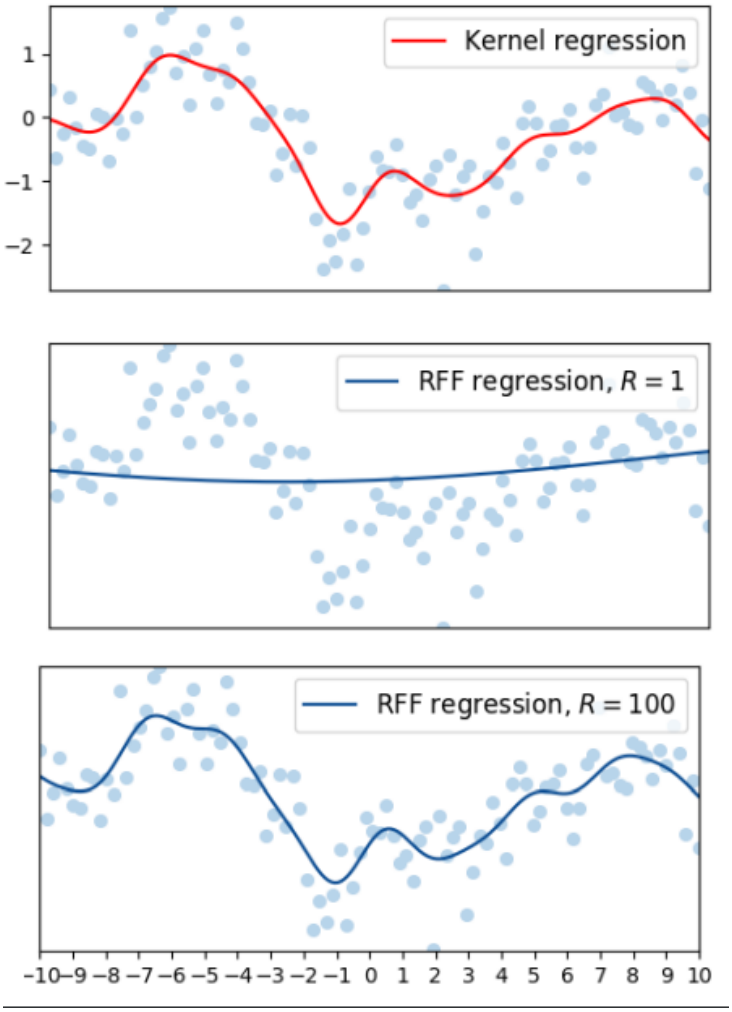



Figure 62: Regression predictions using Random Fourier Features. The approximation quality depends on choosing a sufficient number of random features $R$ relative to the complexity of the underlying function.

**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: $\phi(x_i) = \sqrt{2/R}[\cos(w_1^\top x_i + b_1), \ldots, \cos(w_R^\top x_i + b_R)]^\top$
2. Compute leverage scores in the RFF feature space
3. Sample according to leverage scores
4. Run regression on the sampled points

# 94 Multi-Armed Bandits

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 Bandits: The Setup**

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 step $t$:

1. Choose an action $a_t$ from $K$ available options ("arms")
2. Observe reward $r_t$ (stochastic, depending on which arm was pulled)

**Objective**: Maximise cumulative reward $\sum_{t=1}^{T} r_t$
Unlike full reinforcement learning, there is no state that evolves—each decision is independent (no persistent environment that changes based on your actions).

**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?

## 94.1 Contextual Bandits

**Contextual Bandits**

An extension where the reward depends on observable context:

$$\max_{a_1,\ldots,a_T} \sum_{t=1}^{T} r(a_t, x_t)$$

where $x_t$ is the context observed before choosing action $a_t$.
**Example**: In ad placement, $x_t$ might be the user's browsing history. The best ad to show depends on who is viewing the page.
We must learn the relationship between context, action, and reward—but the model is a **nuisance** rather than the goal. We care only about making good decisions.

## 94.2 The Exploration–Exploitation Trade-off

This is the central challenge in bandit problems:

**Exploration vs Exploitation**

**Exploitation**: Choose the action with the highest estimated reward based on current knowledge. 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 ⇒ you may miss better options you never tried
- Too much exploration ⇒ you waste resources on actions you already know are bad

A good bandit algorithm must balance these competing goals dynamically over time.

## 94.3 Solution 1: $\epsilon$-Greedy

The simplest approach to balancing exploration and exploitation:

**$\epsilon$-Greedy Algorithm**

At each time step:

- With probability $1-\epsilon$: Choose the action with the highest estimated reward (exploit)
- With probability $\epsilon$: Choose an action uniformly at random (explore)

Typically $\epsilon$ is small (e.g., 0.05 or 0.1).

**Advantages**: Extremely simple to implement and understand.
**Disadvantages**:

- Explores uniformly, even among arms that are clearly suboptimal
- The exploration rate $\epsilon$ is constant—we pay the same exploration cost early (when exploration is valuable) and late (when we should mostly exploit)
- No principled way to choose $\epsilon$

## 94.4 Solution 2: Upper Confidence Bound (UCB)

A more intelligent approach that adapts exploration based on uncertainty:

**UCB Algorithm**

At each time step, choose the action that maximises:

$$\bar{r}_a + \sqrt{\frac{2\log t}{n_a}}$$

where:

- $\bar{r}_a$: Average reward observed from action $a$ (exploitation term)
- $n_a$: Number of times action $a$ has been chosen
- $t$: Total number of time steps so far
- $\sqrt{\frac{2\log t}{n_a}}$: Uncertainty bonus (exploration term)

**UCB: "Optimism Under Uncertainty"**

The key insight: if we are uncertain about an arm's true value, give it the benefit of the doubt.
**The uncertainty bonus**:

- Large when $n_a$ is 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

This creates "optimistic" estimates of each arm's value, ensuring underexplored arms are given fair chances.

**UCB Properties**

**Sublinear regret**: The cumulative regret (loss from not always choosing optimally) grows as $O(\sqrt{T\log T})$. This is much better than linear regret (where average performance never improves).
**Adaptive behaviour**:

- **Early**: The uncertainty term dominates, encouraging broad exploration
- **Late**: The reward term dominates, focusing on exploitation
- **Never stops exploring entirely**: The $\log t$ term ensures periodic revisiting of all arms

**Implicit uncertainty quantification**: UCB derives the exploration bonus from concentration inequalities (e.g., Hoeffding's inequality), providing principled confidence bounds.

**Why Sublinear Regret Matters**

**Regret** measures the difference between our cumulative reward and what we would have earned always choosing the best arm:

$$\text{Regret}(T) = T \cdot r^* - \sum_{t=1}^{T} r_t$$

where $r^*$ is the expected reward of the best arm.
**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 make fewer and fewer mistakes over time. Per-round regret $\to 0$ as $T \to \infty$. UCB achieves this because it focuses exploration on promising arms and reduces wasteful exploration over time.

**When UCB may struggle**: UCB assumes a stationary environment where reward distributions do not change. In non-stationary settings, $\epsilon$-greedy (with its constant exploration) may adapt better.

## 94.5 Solution 3: Thompson Sampling

A Bayesian alternative that achieves the same theoretical guarantees as UCB but through a different mechanism:

**Thompson Sampling Algorithm**

Maintain a posterior distribution over each arm's expected reward. At each time step:

1. Sample a value from each arm's posterior distribution
2. Choose the arm with the highest sampled value
3. Observe the reward and update that arm's posterior

Formally, choose action $a$ with probability:

$$p(a) = P(r_a = \max_i r_i)$$

the probability that arm $a$ is optimal.

**Thompson Sampling: Why It Works**

**Automatic exploration**: Arms with uncertain posteriors have high variance. Sometimes they sample high (leading to exploration), sometimes low. Well-understood arms have low variance, so their samples are predictable.
**Probabilistic matching**: Each arm is chosen in proportion to the probability that it is optimal. This is a natural way to balance exploration and exploitation.
**Key advantages**:

- Also achieves sublinear regret (provably as good as UCB)
- More natural Bayesian interpretation
- Adapts well to non-stationary environments (posteriors can incorporate forgetting)
- Often the default choice in modern applications
- Naturally maintains calibrated uncertainty estimates

**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.

# 95 Estimating Prevalence: AIPW

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.

## 95.1 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

## 95.2 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.

## 95.3 The AIPW Estimator

**Augmented Inverse Propensity Weighting**

The AIPW estimator for the population mean of $Y$:

$$\hat{Y}_{\text{AIPW}} = \hat{g}(X) + \frac{R}{\pi(X)}(Y - \hat{g}(X))$$

where:

- $\hat{g}(X)$: Model's prediction of $Y$ given $X$
- $\pi(X)$: Propensity score—the probability that unit with features $X$ was sampled
- $R \in \{0, 1\}$: Indicator for whether the unit was sampled
- $Y$: True label (observed only if $R = 1$)

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.

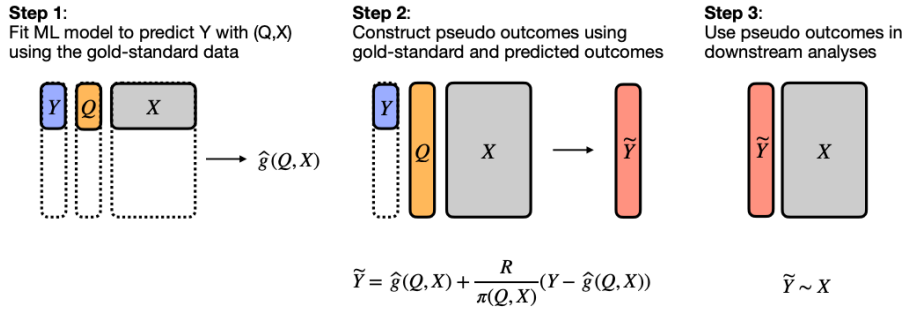



Figure 63: The AIPW workflow: (1) Fit a model on gold-standard labelled data, (2) Construct pseudo-outcomes that combine model predictions with IPW corrections, (3) Use pseudo-outcomes for population inference.

## 95.4 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 the model is perfect** ($\hat{g}(X) = Y$ for all units):

- The residual $Y - \hat{g}(X) = 0$
- The correction term vanishes
- We simply average model predictions across the population (low variance)

**If the model is imperfect**:

- The correction term $\frac{R}{\pi(X)}(Y - \hat{g}(X))$ accounts for model errors
- Labelled examples reveal and correct for systematic biases
- The estimate remains unbiased

We get variance reduction from the model (predictions on all units) AND unbiasedness from the IPW correction (on labelled units).

**Double Robustness**

AIPW has a remarkable property called **double robustness**:
The estimator is consistent (converges to the true value) if **either**:

1. The outcome model $\hat{g}(X)$ is correctly specified, OR
2. The propensity score $\pi(X)$ is correctly specified

You do not need both to be correct—only one. This provides robustness against model misspecification that pure model-based or pure IPW approaches lack.
**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.

**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

# 96 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 trade-off**: Balance learning and earning
   - **$\epsilon$-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 $\to 0$
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

**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

# 97 Overview

In machine learning, we often focus on point predictions—our 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 medical diagnosis system: a prediction of "malignant" with 99% confidence demands different action than the same prediction with 55% confidence.
This week introduces two complementary approaches to quantifying predictive uncertainty, each with distinct strengths and trade-offs.

**Two Approaches to Uncertainty**

1. **Gaussian Processes**: Assume a probabilistic model; get full posterior distributions with calibrated uncertainty. Rich uncertainty quantification, but computationally expensive.
2. **Conformal Inference**: Model-agnostic; get prediction intervals with coverage guarantees. Distribution-free and scalable, but only marginal (not conditional) coverage.

## 97.1 Why Uncertainty Matters

**Motivations for Uncertainty Quantification**

- **Decision-making**: Know when to trust predictions; quantify risk before acting
- **Active learning**: Sample where uncertainty is high to maximise information gain
- **Bayesian optimisation**: Balance exploration (uncertain regions) and exploitation (promising regions)
- **Outlier detection**: Flag predictions with unusually high uncertainty
- **Model selection**: Prefer models with well-calibrated uncertainty estimates

# 98 Gaussian Processes

A Gaussian Process (GP) is a distribution over functions, not just parameters. This is a powerful conceptual shift: instead of finding a single "best" function $\hat{f}$, we maintain a **distribution over all possible functions** consistent with our data and prior beliefs.

**Gaussian Process Definition**

A **Gaussian Process** is a collection of random variables, any finite subset of which has a joint Gaussian distribution.
A GP is fully specified by:

- **Mean function**: $m(x) = \mathbb{E}[f(x)]$ (often set to 0 for simplicity)
- **Covariance function (kernel)**: $k(x, x') = \mathbb{E}[(f(x) - m(x))(f(x') - m(x'))]$

We write: $f \sim \mathcal{GP}(m, k)$

**Intuition**: Given some data points, a GP helps you predict where other points might lie, along with how certain it is about those predictions. Think of it as a very smart, very flexible curve that we fit through data—one that doesn't 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.

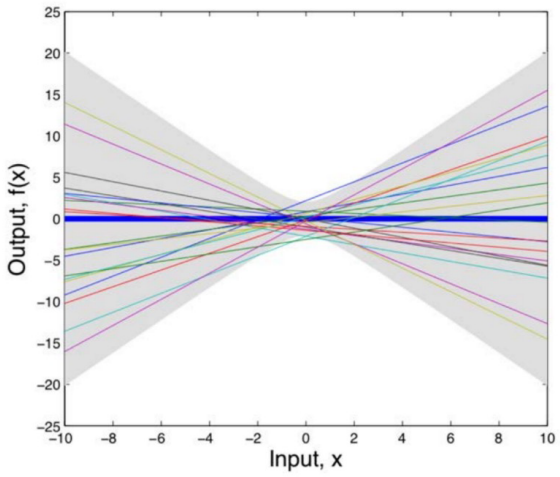



Figure 64: A GP defines a distribution over functions, with the shaded region showing uncertainty. The mean function (solid line) is our best estimate, while the shaded region indicates where the true function might plausibly lie.

## 98.1 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.

**Ridge regression as an example**: Recall from Week 6 that ridge regression can be viewed as placing a Gaussian prior on the coefficients $\beta$. Coefficients near zero are considered more probable than large values, reflecting a prior belief that the true relationship is likely simple. This prior on $\beta$ 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.

## 98.2 Components of a Gaussian Process

### 98.2.1 Mean Function $m(x)$

The mean function describes the average value of the function we are trying to predict at point $x$. It represents our **prior expectation**—what we believe about the function before seeing any data.

- Commonly set to zero, since the kernel provides enough flexibility to model the data
- Setting $m(x) = 0$ is a simplification saying that, before seeing data, we don't prefer one function value over another
- For some applications, we might use a parametric mean function (e.g., linear) to encode prior knowledge

### 98.2.2 Kernel (Covariance Function) $k(x, x')$

The kernel embodies our assumptions about the function we want to learn. It defines the covariance between function values at any two points $x$ and $x'$.
**What the kernel captures:**

- 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—the kernel determines what kinds of functions are probable before seeing data.

**Common Kernel Choices**

- **Squared Exponential (RBF)**: Assumes very smooth functions. Nearby points have similar values; similarity decays exponentially with distance.
- **Periodic**: Assumes the function repeats itself over time. Useful for seasonal data.
- **Linear**: Assumes a linear relationship. Using a linear kernel in a GP reduces to Bayesian linear regression.

The choice of kernel is critical and heavily influences model performance. It determines the shape and smoothness of the functions in our "function space."

## 98.3 Properties of Gaussian Processes

**GP Properties**

- **Non-parametric**: No fixed functional form; defines a distribution over all possible functions in the hypothesis class
- **Prior knowledge incorporation**: Kernel choice encodes assumptions (smoothness, periodicity, etc.)
- **Uncertainty quantification**: Built-in confidence intervals that widen where data is sparse
- **Data-efficient**: Strong priors enable good performance with small datasets
- **Flexibility**: Works well when you believe the data has rich structure, even with limited observations

**NB!**

**The Main Downside: Computational Cost**
GP inference requires inverting an $n \times n$ matrix, which is $O(n^3)$. For large datasets, this becomes prohibitive. Scaling GPs to large datasets is an active research area, with approaches including sparse GPs, inducing points, and stochastic variational inference.

## 98.4 GP Inference: From Prior to Posterior

### 98.4.1 The Process

1. **Prior distribution over functions**: The GP prior encapsulates our beliefs about functions *before* observing any data. This is determined by our choice of mean function and kernel.
2. **Condition on observed data**: We update our beliefs based on data. If we observe $y = f(x) + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \sigma^2)$ (Gaussian noise), we can compute the posterior analytically.
3. **Posterior predictive distribution**: For any new test points, we obtain not just point predictions but a full probability distribution over possible outputs.

**Conditional Normality Assumption**

The fundamental assumption in GPs is that all points are **conditionally normal**: for any set of inputs, the corresponding outputs follow a multivariate normal distribution. This is what makes GP inference tractable—we can use standard results for conditioning multivariate Gaussians.

### 98.4.2 Joint Distribution

**Joint Distribution of Training and Test Points**

Given training data $(X, y)$ and test inputs $X_*$, the joint distribution of observed and predicted values is:

$$\begin{bmatrix} y \\ f_* \end{bmatrix} \sim \mathcal{N}\left( \begin{bmatrix} m(X) \\ m(X_*) \end{bmatrix}, \begin{bmatrix} K_{XX} + \sigma^2 I & K_{X*} \\ K_{*X} & K_{**} \end{bmatrix} \right)$$

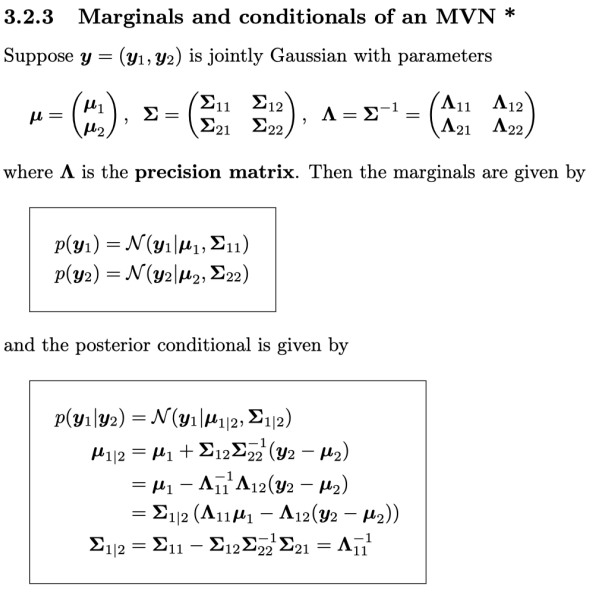
where:

- $y$: observed training labels
- $f_*$: function values at test points (what we want to predict)
- $K_{XX} = k(X, X)$: covariance matrix among training points
- $K_{X*} = k(X, X_*)$: covariance between training and test points
- $K_{*X} = k(X_*, X) = K_{X*}^\top$: covariance between test and training points
- $K_{**} = k(X_*, X_*)$: covariance among test points
- $\sigma^2 I$: observation noise added to training points

**Interpreting the covariance blocks:**

- $K_{XX}$: How training points relate to each other

- $K_{X*}$ and $K_{*X}$: How much the function values at training points inform us about function values at test points
- $K_{**}$: 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



#### 3.2.3 Marginals and conditionals of an MVN *

Suppose $\boldsymbol{y} = (\boldsymbol{y}_1, \boldsymbol{y}_2)$ 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

$$p(\boldsymbol{y}_1) = \mathcal{N}(\boldsymbol{y}_1 | \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_{11})$$
$$p(\boldsymbol{y}_2) = \mathcal{N}(\boldsymbol{y}_2 | \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_{22})$$

and the posterior conditional is given by

$$\begin{aligned} p(\boldsymbol{y}_1 | \boldsymbol{y}_2) &= \mathcal{N}(\boldsymbol{y}_1 | \boldsymbol{\mu}_{1|2}, \boldsymbol{\Sigma}_{1|2}) \\ \boldsymbol{\mu}_{1|2} &= \boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}(\boldsymbol{y}_2 - \boldsymbol{\mu}_2) \\ &= \boldsymbol{\mu}_1 - \boldsymbol{\Lambda}_{11}^{-1}\boldsymbol{\Lambda}_{12}(\boldsymbol{y}_2 - \boldsymbol{\mu}_2) \\ &= \boldsymbol{\Sigma}_{1|2}\left(\boldsymbol{\Lambda}_{11}\boldsymbol{\mu}_1 - \boldsymbol{\Lambda}_{12}(\boldsymbol{y}_2 - \boldsymbol{\mu}_2)\right) \\ \boldsymbol{\Sigma}_{1|2} &= \boldsymbol{\Sigma}_{11} - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21} = \boldsymbol{\Lambda}_{11}^{-1} \end{aligned}$$

Figure 65: Conditioning on observed data transforms the prior into a posterior. The left shows the marginal (prior) distribution; the right shows the conditional (posterior) distribution after observing data.

## 98.5 Predictive Distribution

Conditioning on observed data gives the posterior predictive distribution. This is where the power of GPs becomes apparent: we get both a point prediction *and* a measure of uncertainty.

**GP Posterior Predictive**

$$f_* | X_*, X, y \sim \mathcal{N}(\mu_*, \Sigma_*)$$

**Mean** (point prediction):

$$\mu_* = m(X_*) + K_{*X}(K_{XX} + \sigma^2 I)^{-1}(y - m(X))$$

**Variance** (uncertainty):

$$\Sigma_* = K_{**} - K_{*X}(K_{XX} + \sigma^2 I)^{-1} K_{X*}$$

### 98.5.1 Understanding the Mean $\mu_*$

The posterior mean has an intuitive interpretation. We start with our prior mean $m(X_*)$, then adjust based on how the observed data deviates from prior expectations:

- $m(X_*)$: Prior mean at test points—our baseline expectation before considering training data

- $K_{*X}$: Covariance between test and training inputs. Measures how similar each test point is to each training point (according to the kernel). Points more similar to the test point will have greater influence.
- $(K_{XX} + \sigma^2 I)^{-1}$: Inverse covariance matrix of training points (regularised by noise). Acts as a normalisation factor that weights the influence of each training point based on its relationship to other training points.
- $(y - m(X))$: Residuals—how the observed outputs deviate from the prior mean. This is the "surprise" in the training data.

**Interpreting the Mean Formula**

The posterior mean adjusts the prior mean based on:

1. How similar test points are to training points ($K_{*X}$)
2. How training observations deviate from prior expectations ($y - m(X)$)

The prediction is informed by both prior knowledge (encoded in $m$) and observed data, weighted by similarity between input points.

### 98.5.2 Understanding the Variance $\Sigma_*$

The posterior variance captures our remaining uncertainty after observing the training data:

- $K_{**}$: Prior covariance among test points—our initial uncertainty before seeing data
- $K_{*X}(K_{XX} + \sigma^2 I)^{-1} K_{X*}$: Variance "explained" by the training data. This term measures how much our uncertainty is reduced by having observed the training points.

**Intuitive Step-by-Step for Variance**

1. **Start with prior uncertainty**: Begin with $K_{**}$, our initial uncertainty about function values at test points
2. **Subtract explained variance**: Remove the part that can be explained by the relationship with training data
3. **Account for noise**: The term $(K_{XX} + \sigma^2 I)^{-1}$ prevents overconfidence when observations are noisy
4. **Result**: $\Sigma_*$ is a matrix giving variances (diagonal) and covariances (off-diagonal) of predictions

The diagonal elements give the variance at each test point—higher variance means greater uncertainty. Off-diagonal elements show how uncertainties at different test points are correlated.

## 98.6 Connection to Kernel Ridge Regression

This is where the relationship to Week 6 becomes precise. Recall that kernel ridge regression gives predictions:

$$\hat{y} = K_{*X}(K_{XX} + \lambda I)^{-1} y$$

**NB!**

With a zero mean function, the GP posterior mean is **identical** to Kernel Ridge Regression:

$$\mu_* = K_{*X}(K_{XX} + \sigma^2 I)^{-1} y$$

The regularisation parameter $\lambda$ in KRR corresponds to the noise variance $\sigma^2$ in the GP. **The key difference**: GPs also compute $\Sigma_*$, providing **calibrated uncertainty estimates** rather than just point predictions. In KRR, you get point estimates based on a deterministic function. GPs give you a full probabilistic model.

**What GPs add beyond KRR:**

- The diagonal elements of $\Sigma_*$ 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

## 98.7 Visualising GP Behaviour

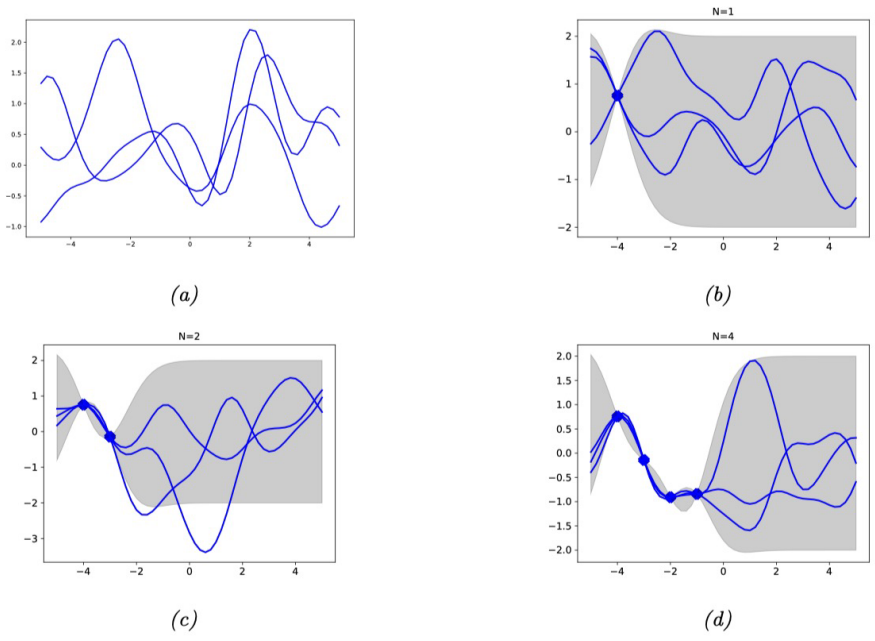



Figure 66: Progression of a GP as data is observed: functions inconsistent with observations are down-weighted. The shaded region represents the confidence interval; sample functions are drawn from the posterior.

**(a) No data ($N = 0$):** Functions sampled from the GP prior are highly varied—high uncertainty about the true function form. The shaded region spans a wide range, reflecting that before seeing any data, many function shapes are plausible.
**(b) One point ($N = 1$):** The confidence interval narrows near the observed point and widens away from it. All sample functions pass through (or near) the observation. Functions inconsistent with this observation are down-weighted.

**(c) Two points ($N = 2$):** Intervals tighten around both observations. Sample functions vary less near data and more in regions without observations. The GP "knows" more where it has seen data.
**(d) Four points ($N = 4$):** Confidence intervals are significantly tighter around all observations. The GP has learned the trend and has much less uncertainty where data is dense.
This illustrates the GP's capability to both interpolate (between data points) and extrapolate (beyond data points), providing a probabilistic framework that naturally incorporates uncertainty.

## 98.8 Variance Properties: Why GPs Excel at Uncertainty

**Good Variance Behaviour**

GP variance increases as test points move away from training data:

- **Near training data**: Low variance (confident predictions)
- **Far from training data**: High variance (uncertain predictions)
- **Rate of increase**: Depends on kernel choice (e.g., length-scale parameter)

This is **exactly what we want**: admit uncertainty where we lack information.

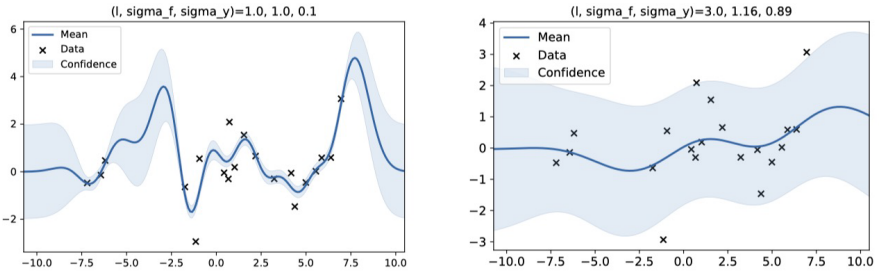



Figure 67: Two GPs with different kernel hyperparameters. Crosses indicate training data. The solid line shows the mean prediction; shaded regions show confidence intervals. Right panel: larger length-scale gives wider confidence bands, indicating that observations influence predictions over a larger range.

**Contrast with other methods:**

- Linear regression and polynomial regression provide point estimates without uncertainty measures
- Even some Bayesian methods don't always show increased variance with distance from training data
- This variance property makes GPs particularly valuable for active learning (sample where uncertain) and Bayesian optimisation (balance exploration and exploitation)

## 98.9 Posterior vs Posterior Predictive

There is an important distinction between two types of uncertainty in GPs:

**Two Types of Uncertainty**

**Posterior of the function** (credible intervals):

- Uncertainty about the true underlying function $f$
- "Where might the true function lie?"
- Computed using $\Sigma_*$ as derived above

**Posterior predictive** (prediction intervals):

- Uncertainty about future observations $y = f(x) + \epsilon$
- Includes observation noise: add $\sigma^2$ to diagonal of $\Sigma_*$
- "Where might future observations fall?"
- Wider than credible intervals because observations include noise

**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.
**Kernel implication**: To shift from posterior of the function to posterior predictive, add the noise variance $\sigma^2$ to the diagonal of the kernel covariance matrix $K_{**}$. This accounts for additional uncertainty from noisy observations when predicting future labels.

## 98.10 Bayesian Optimisation

GPs enable **Bayesian optimisation**—an approach for optimising expensive-to-evaluate functions:

1. Model the objective function with a GP
2. Use an **acquisition function** (e.g., Expected Improvement) to balance exploration and exploitation
3. Sample at the point maximising the acquisition function
4. Update the GP with the new observation; repeat

The GP's uncertainty guides where to sample next:

- **Exploitation**: Sample where predicted values are high (promising regions)
- **Exploration**: Sample where uncertainty is high (regions where we might discover something better)

This is particularly valuable when function evaluations are expensive (e.g., training a neural network, running a physical experiment), as we want to find the optimum with as few evaluations as possible.
**Interactive demo**: http://www.infinitecuriosity.org/vizgp/

### 98.11 GP Summary

**Gaussian Process Takeaways**

GPs are:

1. A probability distribution over functions
2. Updated by conditioning on data to form a posterior
3. Equipped with good variance properties (uncertainty increases away from data)

GPs provide flexible models with extremely well-behaved uncertainty estimates. They are commonly used in Bayesian optimisation and active learning where uncertainty quantification is critical.
**Limitations**: Scaling to large datasets is challenging ($O(n^3)$ complexity) and remains an active research area.

## 99 Conformal Inference

Conformal inference takes a completely different approach to uncertainty: rather than assuming a probabilistic model, it provides **distribution-free** prediction intervals with guaranteed coverage.

**Conformal Inference Goal**

Construct prediction intervals $(l, u)$ such that:

$$P(l \leq y_{\text{new}} \leq u) \geq 1 - \alpha$$

for **any** underlying distribution, using **any** predictive model.

The key insight: we don't need to assume anything about the data distribution or the model's correctness. We only need the data to be **exchangeable** (a weaker condition than i.i.d.).

### 99.1 A Primer on Conformal Inference

Conformal inference creates rigorous prediction intervals that are valid under a specified confidence level, regardless of the underlying distribution of the data. It is based on the idea of **nonconformity measures**—functions that assess how well new observations conform to past observations.
**Key properties:**

- **Distribution-free**: No assumptions about the underlying distribution of the data
- **Model-agnostic**: Can be applied to any predictive model
- **Provably valid**: If you say you're 95% confident the true value lies within the interval, it will indeed lie within the interval 95% of the time in the long run

## 99.2 The Algorithm: Split Conformal Prediction

**Split Conformal Prediction**

1. **Split data**: Divide into training set and calibration set
2. **Fit model** on training set to get $\hat{f}$
3. **Define score function** $s(x, y)$ measuring "nonconformity"
   - Common choice: $s(x, y) = |y - \hat{f}(x)|$ (absolute residual)
   - Larger scores indicate worse fit (more "nonconforming")
4. **Compute scores** on calibration set: $s_1, \ldots, s_n$
5. **Find quantile**: $\hat{q}$ = the $\lceil (n+1)(1-\alpha) \rceil / n$ quantile of scores
6. **Prediction set**: $\mathcal{C}(x_{\text{new}}) = \{y : s(x_{\text{new}}, y) \leq \hat{q}\}$

For regression with absolute residual scores, the prediction set simplifies to an interval:

$$\mathcal{C}(x_{\text{new}}) = \hat{f}(x_{\text{new}}) \pm \hat{q}$$

## 99.3 Why Conformal Inference Works

**Why It Works**

Under exchangeability (weaker than i.i.d.), the calibration scores and the new point's score are exchangeable.
The new score has probability $\leq \alpha$ of exceeding the $(1-\alpha)$ quantile of the calibration scores.
**No assumptions** about the model or data distribution—only exchangeability.

The validity guarantee comes from a simple counting argument: if all $n+1$ scores (calibration plus new) are exchangeable, then the new score is equally likely to be in any rank position. The probability it exceeds the $(1-\alpha)$ quantile is at most $\alpha$.

## 99.4 Step-by-Step Process

**Set target coverage probability**:

- Choose significance level $\alpha$ (e.g., $\alpha = 0.05$ for 95% coverage)
- Goal: prediction interval $(l, u)$ such that $P(l \leq y_{\text{new}} \leq u) \geq 1 - \alpha$

**Calculate prediction interval**:

1. **Define heuristic notion of uncertainty**: Identify a measure reflecting confidence in predictions
2. **Formalise as score function** $s(x, y)$: Assigns a numerical value indicating how far the predicted value $\hat{f}(x)$ is from the true value $y$
   - $s(x, y) = |y - \hat{f}(x)|$ is a common choice
   - Larger score = worse prediction
3. **Compute quantile** $\hat{q}$: Calculate the $\lceil (n+1)(1-\alpha) \rceil / n$ quantile of scores on calibration data

- This quantile represents the threshold that a certain proportion of scores fall below

4. **Form prediction set** $\mathcal{C}(X_{\text{test}})$: Include any value $y$ where $s(X_{\text{test}}, y) \leq \hat{q}$

## 99.5 Example: Non-Normal Errors

Consider a data generating process $y = \beta X + \epsilon$, where $\epsilon$ has a highly non-normal distribution (e.g., heavy tails, skewness).
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 $\epsilon$'s distribution. It may not be *tight* (optimal width), but it is valid.

## 99.6 Handling Heteroskedasticity

If variance changes with $X$ (heteroskedasticity), use a **normalised** score:

$$s(x, y) = \frac{|y - \hat{f}(x)|}{\hat{\sigma}(x)}$$

where $\hat{\sigma}(x)$ is a model of the residual standard deviation at $x$.
**Implementation:**

1. Train a model to predict $\hat{f}(x)$
2. Train a separate model to predict $\hat{\sigma}(x)$ (e.g., using absolute residuals from the first model)
3. Use normalised scores for calibration and prediction

This gives tighter intervals where variance is low and wider intervals where variance is high, while maintaining the coverage guarantee.

## 99.7 Marginal vs Conditional Coverage

**NB!**

Conformal inference guarantees **marginal coverage**:

$$P(y_{\text{new}} \in \mathcal{C}(x_{\text{new}})) \geq 1 - \alpha$$

averaged over all $x$. It does **not** guarantee **conditional coverage**:

$$P(y_{\text{new}} \in \mathcal{C}(x_{\text{new}}) | x_{\text{new}} = x) \geq 1 - \alpha \quad \forall x$$

**Marginal coverage example**: The coverage guarantee is averaged over all points:

| $X$ | Prediction | Interval | Coverage |
|---|---|---|---|
| 1 | 0.1 | $(0, 0.2)$ | 99% |
| 2 | 0.2 | $(0.1, 0.3)$ | 90% |
| 3 | 0.3 | $(0.2, 0.4)$ | 81% |
| Overall (marginal): | | | 90% |

Even though individual $X$ values have different coverage rates (some much higher, some lower than 90%), the overall coverage meets the target.
**Conditional coverage**: This would require the coverage guarantee to hold **within each value of $X$**:

| $X$ | Prediction | Interval | Coverage |
|---|---|---|---|
| 1 | 0.1 | $(0.02, 0.18)$ | 90% |
| 2 | 0.2 | $(0.1, 0.3)$ | 90% |
| 3 | 0.3 | $(0.18, 0.42)$ | 90% |
| Overall: | | | 90% |

**The challenge**: Achieving conditional coverage requires intervals that adapt to the distribution of $y$ at each $x$ value, accounting for all potential variations. There isn't a straightforward method to achieve exact conditional coverage in all cases—this remains an active research area.

## 100 Comparison: GPs vs Conformal Inference

**GPs vs Conformal Inference**

| | **Gaussian Processes** | **Conformal Inference** |
|---|---|---|
| Assumptions | Gaussian, kernel choice | Exchangeability only |
| Output | Full posterior distribution | Prediction intervals |
| Calibration | Automatic (Bayesian) | Requires calibration set |
| Scalability | $O(n^3)$ | $O(n)$ |
| Model-agnostic | No (GP-specific) | Yes (any model) |
| Coverage guarantee | Conditional (if model correct) | Marginal |
| Uncertainty type | Mean + variance | Interval only |

**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

# 101 Summary

**Key Concepts from Week 9**

1. **Gaussian Processes**: Distributions over functions; posterior mean equals KRR, but also provides calibrated variance
2. **GP inference**: Condition a joint Gaussian on observed data; variance increases away from training points
3. **GP-KRR connection**: With zero mean function, GP posterior mean is identical to kernel ridge regression; GPs extend this by also computing uncertainty
4. **Posterior vs predictive**: Function uncertainty (credible intervals) vs observation uncertainty (prediction intervals, which include noise)
5. **Conformal inference**: Distribution-free prediction intervals with marginal coverage guarantees
6. **Score function**: Defines what "nonconformity" means; quantile of scores gives interval width
7. **Marginal vs conditional**: Conformal gives marginal coverage (averaged over $X$), not conditional (for each $X$)
8. **Tradeoff**: GPs give richer uncertainty but scale poorly ($O(n^3)$); conformal is scalable ($O(n)$) but only provides marginal coverage

# 102 Overview

This week introduces the foundational concepts of neural networks, beginning with the perceptron and building toward multi-layer architectures. Neural networks represent a paradigm shift from the models we have studied so far: rather than hand-crafting feature transformations, we learn them directly from data.

**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 key insight is that by stacking layers of simple transformations with non-linearities, we can approximate arbitrarily complex functions—and crucially, we can learn the parameters of all these transformations end-to-end using gradient descent.

# 103 Perceptrons

The perceptron, originally proposed as a model of biological neuron function, is the simplest neural network architecture. It serves as both a historical starting point and a conceptual foundation for understanding more complex networks.

## 103.1 The Algorithm

The perceptron is a **binary linear classifier**—a non-probabilistic version of logistic regression. It maps inputs $\mathbf{x}$ (a feature vector) to a binary output based on a linear prediction function.

**Perceptron Definition**

**Decision function:**

$$f(\mathbf{x}; \boldsymbol{\beta}) = \mathbb{I}(\mathbf{x}^\top \boldsymbol{\beta} \geq 0) \tag{50}$$

where $\mathbb{I}(\cdot)$ is the **indicator function** (Heaviside step function) that outputs 1 if the argument is true, and 0 otherwise.

**Interpretation:** The perceptron computes a weighted sum of inputs $\mathbf{x}^\top \boldsymbol{\beta}$ and applies a hard threshold at zero. Points on one side of the hyperplane defined by $\mathbf{x}^\top \boldsymbol{\beta} = 0$ are classified as class 1; points on the other side as class 0.

The hard threshold is what makes the perceptron non-probabilistic: unlike logistic regression, which outputs probabilities via the sigmoid function, the perceptron commits to a discrete classification.

### 103.1.1 The Perceptron Learning Rule

The perceptron learns by iteratively correcting misclassifications:

**Perceptron Update Rule**

For a misclassified training point $(\mathbf{x}_j, y_j)$:

$$\beta_i(t+1) = \beta_i(t) + r \cdot (y_j - \hat{f}_j(t)) \cdot x_{j,i} \tag{51}$$

where:

- $\beta_i(t)$ is the $i$-th coefficient at iteration $t$
- $r$ is the **learning rate** (hyperparameter)
- $y_j - \hat{f}_j(t)$ is the **error** (non-zero only for misclassifications)
- $x_{j,i}$ is the $i$-th feature of the misclassified point

**Process:**

1. Compute output: $\hat{f}_j(t) = \mathbb{I}(\mathbf{x}_j^\top \boldsymbol{\beta}(t) \geq 0)$
2. If misclassified, update: $\boldsymbol{\beta}(t+1) = \boldsymbol{\beta}(t) + r(y_j - \hat{f}_j(t))\mathbf{x}_j$
3. Repeat until convergence or maximum iterations reached

The intuition is straightforward: when a point is misclassified, we adjust the weight vector in the direction of the misclassified point's features. If $y_j = 1$ but we predicted 0, we add $r \cdot \mathbf{x}_j$ to $\boldsymbol{\beta}$, nudging the decision boundary toward classifying $\mathbf{x}_j$ correctly.

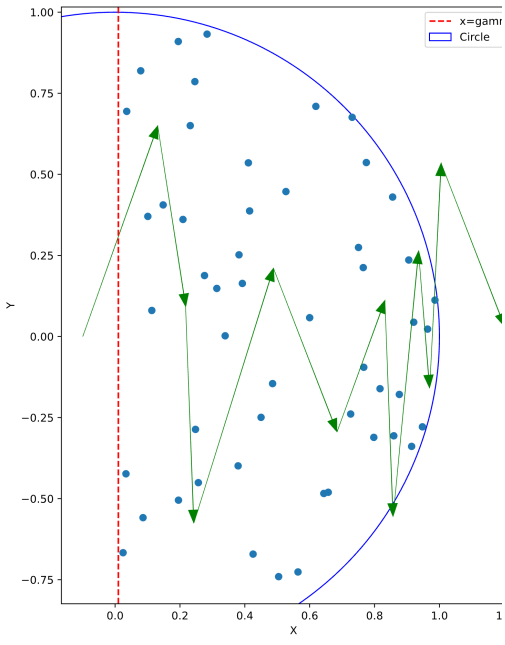



Figure 68: Perceptron learning: the decision boundary (dashed line) adjusts iteratively as misclassifications are corrected. The algorithm bounces around but generally moves toward a separating hyperplane.

### 103.1.2 Connection to Logistic Regression

The perceptron update rule is reminiscent of gradient descent for logistic regression. In logistic regression, the gradient of the negative log-likelihood with respect to $\boldsymbol{\beta}$ is:

$$\nabla_{\boldsymbol{\beta}} \text{NLL}(\boldsymbol{\beta}) = -\frac{1}{n} \sum_{j=1}^{n} (y_j - \hat{p}_j) \cdot \mathbf{x}_j \tag{52}$$

where $\hat{p}_j = \sigma(\mathbf{x}_j^\top \boldsymbol{\beta})$ is the predicted probability.

**Perceptron vs Logistic Regression**

**Similarities:**

- Both update weights based on prediction errors
- Both move weights in the direction of misclassified features

**Differences:**

- **Perceptron**: Updates for single observations, uses hard threshold, suitable only for linearly separable data
- **Logistic regression**: Updates based on entire dataset (batch), uses probabilistic predictions, works for non-separable data via soft boundaries

## 103.2 Limitations of the Perceptron

The perceptron converges (finds a separating hyperplane) *if and only if* the data is linearly separable. This leads to two fundamental problems.

### 103.2.1 Linear Separability Requirement

The perceptron can only learn linearly separable functions. The classic counterexample is the XOR function.

**The XOR Problem**

The XOR (exclusive OR) function outputs 1 when exactly one input is 1:

| $x_1$ | $x_2$ | XOR |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

No single straight line can separate the $(0,0), (1,1)$ points (class 0) from the $(0,1), (1,0)$ points (class 1).

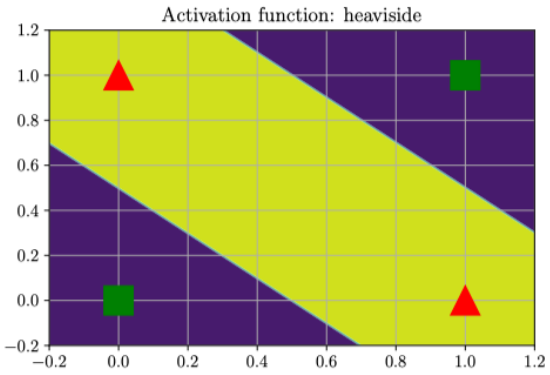



Figure 69: XOR is not linearly separable—no single hyperplane can separate the classes. The perceptron will cycle indefinitely without converging.

When data is not linearly separable, the perceptron algorithm will continue updating weights indefinitely without ever correctly classifying all points.

### 103.2.2 No Margin Maximisation

Even when data *is* linearly separable, infinitely many hyperplanes correctly classify all points. The perceptron has no mechanism to prefer one over another.

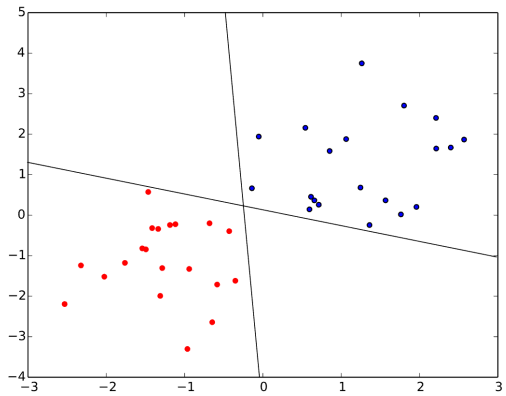



Figure 70: Multiple valid decision boundaries exist for linearly separable data. The perceptron finds *some* separating hyperplane, but not necessarily the best one.

Ideally, we would choose the hyperplane that maximises the margin between classes for better generalisation. This is precisely what Support Vector Machines (SVMs) do, but the perceptron lacks this capability.

**NB!**

[Historical Context: The First AI Winter] These limitations, particularly XOR, were publicised by Minsky and Papert in their influential 1969 book *Perceptrons*. Their analysis led many researchers to abandon neural networks, triggering the first "AI winter."
Interest revived in the 1980s and 1990s with two key developments:

1. **Multi-layer networks**: Adding hidden layers enables learning non-linear decision boundaries
2. **Backpropagation**: An efficient algorithm for training multi-layer networks

The realisation that depth plus non-linearity could solve XOR (and much more) reignited the field.

# 104 Feed-Forward Neural Networks

The single-layer perceptron can be seen as the simplest feed-forward neural network—one with no hidden layers. To overcome its limitations, we add layers and non-linearities.

## 104.1 From Fixed to Learned Features

In the models we have studied so far, feature engineering was explicit: we chose basis functions, kernels, or polynomial expansions based on domain knowledge.

**The Paradigm Shift**

**Traditional approach (fixed features):**

$$f(\mathbf{x};\boldsymbol{\theta}) = \boldsymbol{\theta}^\top \phi(\mathbf{x}) \tag{53}$$

where $\phi(\mathbf{x})$ is a *fixed*, hand-crafted feature transformation (e.g., polynomial features, RBF kernel features).
**Neural network approach (learned features):**

$$f(\mathbf{x};\boldsymbol{\theta}_1,\boldsymbol{\theta}_2) = \boldsymbol{\theta}_2^\top \phi(\mathbf{x};\boldsymbol{\theta}_1) \tag{54}$$

where $\phi$ is *learned from data*—the feature transformation itself has parameters that we optimise.
**Deep networks** compose multiple layers:

$$f(\mathbf{x};\boldsymbol{\theta}) = f_L(f_{L-1}(\cdots f_1(\mathbf{x}))) \tag{55}$$

where each $f_l(\mathbf{x}) = \phi(\boldsymbol{\theta}_l \cdot \mathbf{x})$ applies a linear transformation followed by a non-linear activation.

This is the key insight of deep learning: rather than hand-crafting features, we learn a hierarchy of increasingly abstract representations. Lower layers might detect simple patterns (edges in images, phonemes in speech); higher layers combine these into complex concepts (objects, words).

## 104.2 Hierarchical Feature Learning

The feed-forward architecture is built on the premise that complex representations can be learned by composing simpler ones. Each layer's output serves as the input to the next layer:

$$f(\mathbf{x}; \boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_L) = f_L(f_{L-1}(\cdots f_2(f_1(\mathbf{x}))))) \tag{56}$$

where $f_l(\mathbf{x}) = \sigma(\mathbf{W}_l\mathbf{x} + \mathbf{b}_l)$ typically consists of:

- A **linear transformation**: $\mathbf{z} = \mathbf{W}_l\mathbf{x} + \mathbf{b}_l$
- A **non-linear activation**: $\mathbf{a} = \sigma(\mathbf{z})$

The ability to learn $\phi$ (the transformation at each layer) is crucial. In traditional machine learning, feature transformation is often manual, based on domain knowledge. With neural networks, these transformations emerge from data. Each layer acts as a feature extractor, transforming data into a more abstract representation useful for the task at hand.

### 104.3 Why Non-Linearity is Essential

Without non-linear activations, depth provides no benefit whatsoever.

**NB!**

[The Collapse of Linear Stacks] Consider a network with only linear layers:

$$f(\mathbf{x}) = \boldsymbol{\theta}_L\boldsymbol{\theta}_{L-1}\cdots\boldsymbol{\theta}_1\mathbf{x} = \mathbf{M}\mathbf{x} \tag{57}$$

where $\mathbf{M} = \boldsymbol{\theta}_L\boldsymbol{\theta}_{L-1}\cdots\boldsymbol{\theta}_1$ is simply another matrix.
No matter how many layers we stack, the result is equivalent to a single linear transformation. Depth provides **no additional representational power** without non-linearity.

**Dimensional Analysis**

Suppose:

- $\boldsymbol{\theta}_1$ is $n_1 \times d$ (maps $d$ inputs to $n_1$ hidden units)
- $\boldsymbol{\theta}_2$ is $n_2 \times n_1$ (maps $n_1$ to $n_2$ units)
- $\vdots$
- $\boldsymbol{\theta}_L$ is $m \times n_{L-1}$ (maps to $m$ outputs)

The product $\boldsymbol{\theta}_L\boldsymbol{\theta}_{L-1}\cdots\boldsymbol{\theta}_1$ is an $m \times d$ matrix—equivalent to a single linear layer with $m \times d$ parameters. All the intermediate structure collapses.
**Consequence:** You cannot encode more information in a stack of linear layers than fits in an $m \times d$ matrix, regardless of how many layers or hidden units you use.

Non-linear activation functions between layers prevent this collapse, allowing the network to represent complex, non-linear relationships.

## 105 Activation Functions

Activation functions introduce non-linearity between layers. They are typically applied element-wise after a linear transformation, creating alternating linear/non-linear layers.

**Role of Activation Functions**

- **Linear layers**: Where we learn parameters (weights and biases)
- **Non-linear activations**: Prevent collapse into a single linear model

The activation function itself typically has no learnable parameters—it is a fixed non-linear transformation.

## 105.1 Common Activation Functions

**Activation Function Definitions**

**Step function** (original perceptron):

$$f(x) = \mathbb{I}(x \geq 0) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{58}$$

Binary output; not differentiable at $x = 0$, so unusable for gradient-based optimisation.
**Sigmoid (logistic)**:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{59}$$

Squashes output to $(0, 1)$; interpretable as probability. Smooth and differentiable everywhere. A nice property: non-parameterised, so nothing needs to be learned.
**Hyperbolic tangent**:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1 \tag{60}$$

Squashes output to $(-1, 1)$; zero-centred, which can help optimisation.
**ReLU** (Rectified Linear Unit):

$$\text{ReLU}(x) = \max(0, x) \tag{61}$$

Simple, computationally efficient, non-saturating for positive inputs. The default choice in modern networks.
**Leaky ReLU**:

$$\text{LeakyReLU}(x) = \max(\alpha x, x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases} \tag{62}$$

where $\alpha$ is small (e.g., 0.01). Prevents "dead neurons" by allowing small gradients for negative inputs.
**Swish**:

$$\text{Swish}(x) = x \cdot \sigma(x) \tag{63}$$

Smooth, non-monotonic; sometimes outperforms ReLU in deep networks. Has the property that $\text{Swish}(x) \approx x$ for large positive $x$ and $\text{Swish}(x) \approx 0$ for large negative $x$.

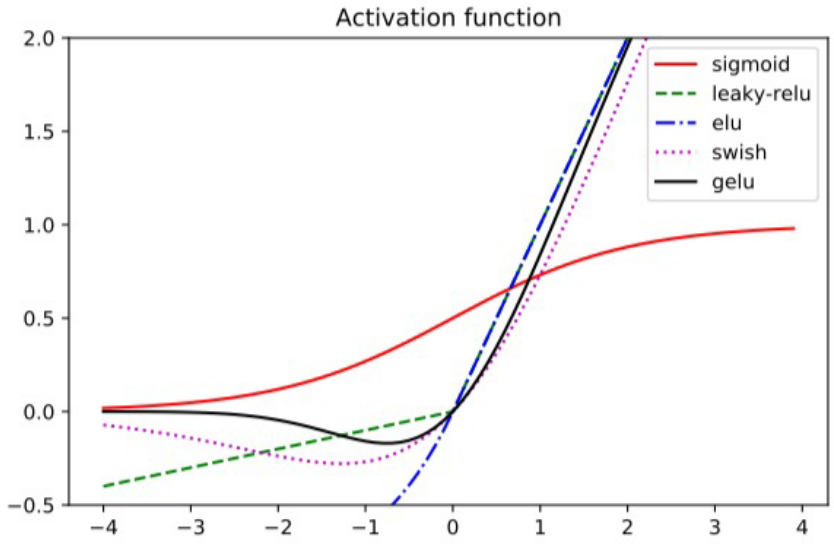



Figure 71: Comparison of common activation functions. Note how sigmoid and tanh saturate (flatten) for large inputs, while ReLU and its variants do not.

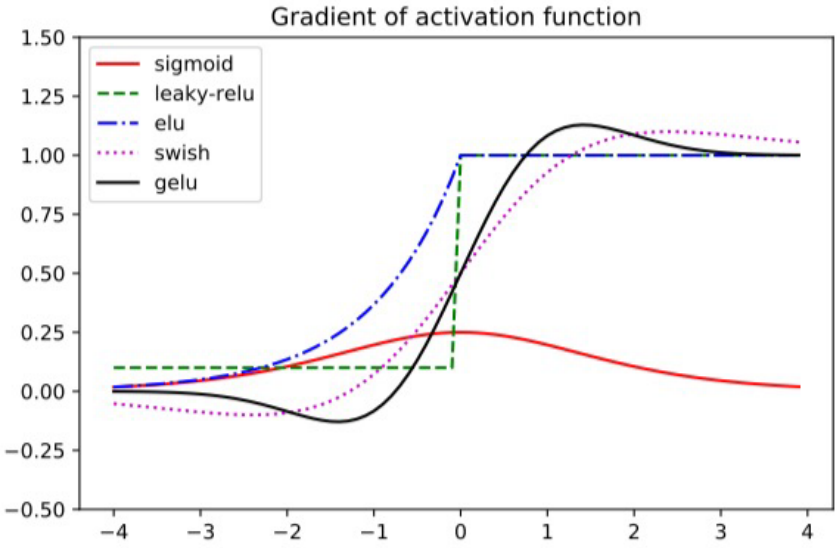



Figure 72: Activation function derivatives—crucial for gradient flow during backpropagation. Sigmoid derivatives vanish for large inputs; ReLU derivatives are constant for positive inputs.

## 105.2 Softmax for Multi-Class Classification

For multi-class classification in the output layer, we use the **softmax** function:

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{64}$$

This transforms a vector of $K$ real numbers into a probability distribution—all outputs are positive and sum to 1. For binary classification ($K = 2$), softmax reduces to the sigmoid function.

# 106 Training Neural Networks

Training a neural network means finding parameters $\boldsymbol{\theta}$ that minimise a loss function $\mathcal{L}$. This is done via gradient-based optimisation, but the complexity of neural networks (many nested functions) makes gradient computation challenging.

## 106.1 Stochastic Gradient Descent

**SGD Update Rule**

$$\boldsymbol{\theta}(t+1) = \boldsymbol{\theta}(t) - \frac{\eta_t}{B} \sum_{b=1}^{B} \nabla_{\boldsymbol{\theta}} \ell(y_b, f(\mathbf{x}_b; \boldsymbol{\theta})) \tag{65}$$

where:

- $\eta_t$ is the **learning rate** at iteration $t$
- $B$ is the **mini-batch size**
- $\nabla_{\boldsymbol{\theta}} \ell$ is the gradient of the loss with respect to parameters

**Mini-batch gradient descent** uses small random subsets of data rather than:

- The entire dataset (batch gradient descent)—computationally expensive, requires all data in memory
- Single examples (pure SGD)—high variance in gradient estimates, noisy updates

Mini-batches provide a balance: reduced variance compared to single-example updates, while remaining computationally tractable. Typical batch sizes range from 32 to 512.

## 106.2 The Credit Assignment Problem

The challenge in neural networks is computing $\nabla_{\boldsymbol{\theta}} \mathcal{L}$—the gradient of the loss with respect to all parameters. With potentially millions of parameters spread across many layers, it is not immediately clear how each parameter contributes to the final loss.
This is the **credit assignment problem**: determining how much "credit" or "blame" each parameter deserves for the network's output error.
The solution lies in systematic application of the chain rule: **backpropagation**.

# 107 Backpropagation

Backpropagation is an efficient algorithm for computing gradients in neural networks. It exploits the compositional structure of neural networks and the chain rule of calculus.

## 107.1 Function Composition and the Chain Rule

A neural network is a composition of functions:

$$\mathbf{o} = f_L \circ f_{L-1} \circ \cdots \circ f_1(\mathbf{x}) \tag{66}$$

where $f_i$ represents layer $i$'s transformation and $\mathbf{o}$ is the output.
Each layer typically maps from one dimension to another:

- $f_1 : \mathbb{R}^d \to \mathbb{R}^{n_1}$ (input to first hidden layer)

- $f_l : \mathbb{R}^{n_{l-1}} \to \mathbb{R}^{n_l}$ (hidden layer $l-1$ to $l$)
- $f_L : \mathbb{R}^{n_{L-1}} \to \mathbb{R}^m$ (final hidden layer to output)

**Chain Rule for Composed Functions**

For the gradient of the output with respect to the input:

$$\frac{\partial \mathbf{o}}{\partial \mathbf{x}} = \mathbf{J}_{f_L} \cdot \mathbf{J}_{f_{L-1}} \cdots \mathbf{J}_{f_1} \tag{67}$$

where $\mathbf{J}_{f_i}$ is the **Jacobian matrix** of layer $i$—the matrix of all partial derivatives of $f_i$'s outputs with respect to its inputs.
The Jacobian $\mathbf{J}_{f_i} \in \mathbb{R}^{n_i \times n_{i-1}}$ has entries:

$$[\mathbf{J}_{f_i}]_{jk} = \frac{\partial [f_i(\mathbf{x})]_j}{\partial x_k} \tag{68}$$

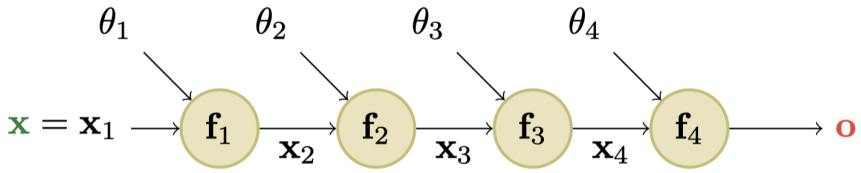



Figure 73: Backpropagation computes gradients layer by layer using the chain rule. The fourth function shown is the loss function.

## 107.2 The Two-Pass Algorithm

Backpropagation consists of two passes through the network:

**Forward and Backward Passes**

**Forward pass:**

1. Propagate inputs through the network, layer by layer
2. Store intermediate activations at each layer (needed for backward pass)
3. Compute the loss at the output

**Backward pass:**

1. Start with the gradient of the loss with respect to the output
2. Propagate gradients backward through each layer using the chain rule
3. At each layer, compute gradients with respect to both parameters and inputs
4. The gradient with respect to inputs becomes the incoming gradient for the previous layer

## 107.3 Backpropagation for an MLP

Consider a simple MLP with:

- Input $\mathbf{x}$
- Linear layer: $\mathbf{x}_2 = \mathbf{W}_1\mathbf{x}$
- Non-linear activation: $\mathbf{x}_3 = \phi(\mathbf{x}_2)$
- Linear layer: $\mathbf{x}_4 = \mathbf{W}_2\mathbf{x}_3$
- Loss: $\mathcal{L}(\mathbf{x}_4, \mathbf{y})$

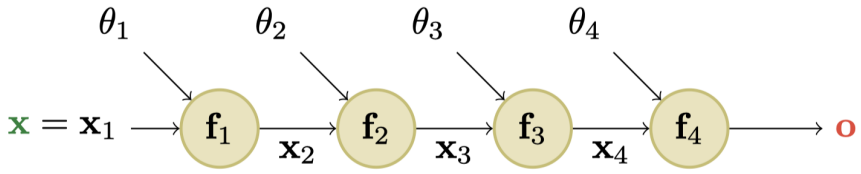



Figure 74: MLP structure with alternating linear and non-linear layers. The final layer is the loss function.

**Gradient Computation via Chain Rule**

The gradients for each layer's parameters are computed by chaining partial derivatives:
**For the output layer parameters $\boldsymbol{\theta}_3$:**

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_3} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_4} \cdot \frac{\partial \mathbf{x}_4}{\partial \boldsymbol{\theta}_3} \tag{69}$$

**For the activation layer parameters $\boldsymbol{\theta}_2$ (if any):**

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_2} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_4} \cdot \frac{\partial \mathbf{x}_4}{\partial \mathbf{x}_3} \cdot \frac{\partial \mathbf{x}_3}{\partial \boldsymbol{\theta}_2} \tag{70}$$

**For the input layer parameters $\boldsymbol{\theta}_1$:**

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_4} \cdot \frac{\partial \mathbf{x}_4}{\partial \mathbf{x}_3} \cdot \frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_2} \cdot \frac{\partial \mathbf{x}_2}{\partial \boldsymbol{\theta}_1} \tag{71}$$

**Key observation:** The coloured terms are *reused* across layers. The backward pass computes these once and propagates them, avoiding redundant computation.

**Algorithm 13.3:** Backpropagation for an MLP with $K$ layers

// Forward pass
$\boldsymbol{x}_1 := \boldsymbol{x}$
**for** $k = 1 : K$ **do**
  $\boldsymbol{x}_{k+1} = \boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{\theta}_k)$ ← Calculate value at each layer
// Backward pass
$\boldsymbol{u}_{K+1} := 1$ ← Initialize gradient
**for** $k = K : 1$ **do**
  $\boldsymbol{g}_k := \boldsymbol{u}_{k+1}^\top \frac{\partial \boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{\theta}_k)}{\partial \boldsymbol{\theta}_k}$ ← Derivative of layer output with respect to parameters
  $\boldsymbol{u}_k^\top := \boldsymbol{u}_{k+1}^\top \frac{\partial \boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{\theta}_k)}{\partial \boldsymbol{x}_k}$ ← Derivative of layer output with respect to input
// Output
Return $\mathcal{L} = \boldsymbol{x}_{K+1}$, $\nabla_{\boldsymbol{x}}\mathcal{L} = \boldsymbol{u}_1$, $\{\nabla_{\boldsymbol{\theta}_k}\mathcal{L} = \boldsymbol{g}_k : k = 1 : K\}$

Loss | Gradient wrt features | Gradient of each layer

Figure 75: Backpropagation algorithm: gradients are computed iteratively from output to input, reusing computations (dynamic programming).

The efficiency of backpropagation comes from this reuse: we compute the gradient signal once and propagate it backward, accumulating the contribution of each layer. This is sometimes called "reverse-mode automatic differentiation."

## 107.4 Computing Individual Layer Derivatives

We still need to know the derivative of each layer type. There are two approaches:

### 107.4.1 Numerical Approximation

The derivative can be approximated using finite differences:

$$\frac{df}{dx} \approx \frac{f(x+h) - f(x)}{h} \tag{72}$$

for small $h$.

**NB!**

[Problems with Numerical Differentiation]

- **Computational cost**: Requires two function evaluations per parameter
- **Choice of $h$**: Too large gives poor approximation; too small causes numerical instability (floating-point errors)
- **No structural advantage**: Does not exploit the layered structure of neural networks

Numerical differentiation is used primarily for *gradient checking*—verifying that analytical gradients are correct—not for training.

### 107.4.2 Automatic Differentiation

Modern deep learning frameworks (PyTorch, TensorFlow, JAX) use **automatic differentiation** (autodiff). This decomposes complex functions into elementary operations with known derivatives, then applies the chain rule automatically.

**Automatic Differentiation Rules**

Autodiff systems construct a computational graph where nodes represent operations and edges represent data flow. Derivatives are computed using:

**Addition rule:**

$$\nabla_{\mathbf{x}}(f(\mathbf{x}) + g(\mathbf{x})) = \nabla_{\mathbf{x}} f(\mathbf{x}) + \nabla_{\mathbf{x}} g(\mathbf{x}) \tag{73}$$

**Product rule:**

$$\nabla_{\mathbf{x}}(f(\mathbf{x}) \cdot g(\mathbf{x})) = f(\mathbf{x}) \cdot \nabla_{\mathbf{x}} g(\mathbf{x}) + g(\mathbf{x}) \cdot \nabla_{\mathbf{x}} f(\mathbf{x}) \tag{74}$$

**Chain rule (composition):**

$$\nabla_{\mathbf{x}} f(g(\mathbf{x})) = \nabla_{\mathbf{u}} f(\mathbf{u})\big|_{\mathbf{u}=g(\mathbf{x})} \cdot \nabla_{\mathbf{x}} g(\mathbf{x}) \tag{75}$$

**Why Deep Learning Frameworks?**

While NumPy supports mathematical operations, it does not support automatic differentiation. Full frameworks like PyTorch, TensorFlow, and JAX:

- Build dynamic computation graphs during the forward pass
- Traverse graphs in reverse to compute gradients via backpropagation
- Handle GPU acceleration and distributed computing

JAX provides autodiff through `jax.grad` and related functions, transforming Python/NumPy code into differentiable form.

## 108 MLP Design

Designing an MLP involves three main choices: architecture, loss function, and optimiser.

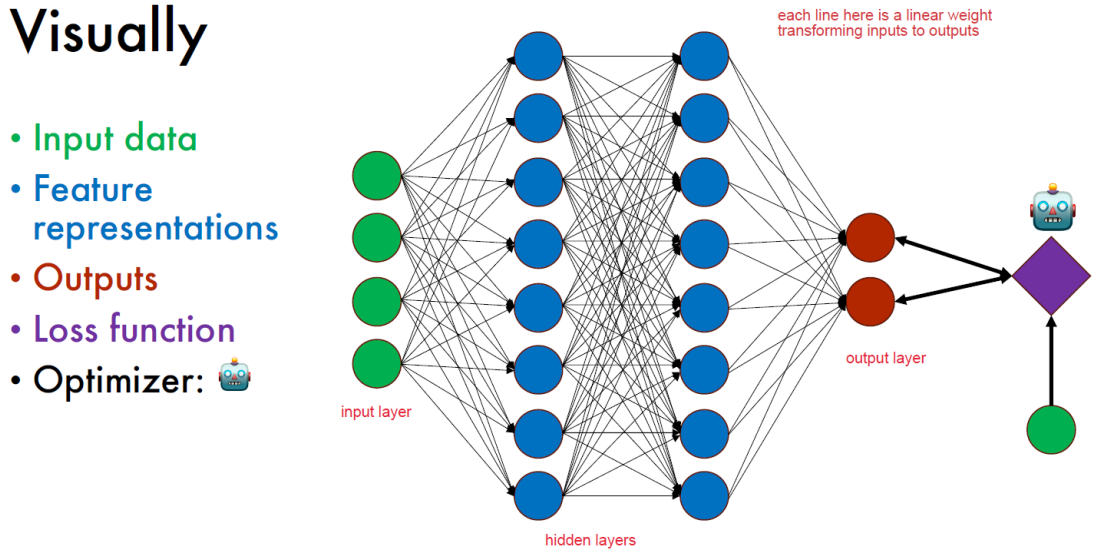



Figure 76: MLP design recipe: architecture, loss function, and optimiser choices.

## 108.1 Architecture Design

**Architecture Choices**

**Number of layers (depth):**

- **Single-layer perceptron**: No hidden layers; only suitable for linearly separable problems
- **One hidden layer**: Can approximate any continuous function (universal approximation), handles non-linear data
- **Multiple hidden layers (deep learning)**: More expressive, can learn hierarchical representations, but harder to train

**Units per layer (width):**

- More units increase model capacity but risk overfitting
- Input layer size matches data dimensionality
- Output layer size matches task: 1 for regression, $K$ for $K$-class classification
- Hidden layer sizes often determined empirically; common patterns include pyramidal (decreasing) or constant width

**Activation functions:**

- ReLU is the default for hidden layers
- Sigmoid/softmax for output layer probabilities
- Tanh sometimes used when zero-centred activations help

## 108.2 A Simple MLP Example

Consider a single hidden layer MLP with $k$ hidden units:

**Single Hidden Layer Architecture**

**Hidden layer $f_1$:**

$$f_1(\mathbf{x};\boldsymbol{\theta}_1) = [\sigma(\boldsymbol{\theta}_1^{(1)\top}\mathbf{x}), \sigma(\boldsymbol{\theta}_1^{(2)\top}\mathbf{x}), \ldots, \sigma(\boldsymbol{\theta}_1^{(k)\top}\mathbf{x})] \tag{76}$$

- Input: $\mathbf{x} \in \mathbb{R}^d$
- Parameters: $\boldsymbol{\theta}_1 \in \mathbb{R}^{d\times k}$ (weight matrix)
- Output: $k$-dimensional vector of learned features
- Role: Learns a non-linear feature representation of the input

**Output layer $f_2$:**

$$f_2(\mathbf{h};\boldsymbol{\theta}_2) = \boldsymbol{\theta}_2^\top \mathbf{h} \tag{77}$$

- Input: $\mathbf{h} \in \mathbb{R}^k$ (hidden layer output)
- Parameters: $\boldsymbol{\theta}_2 \in \mathbb{R}^{k\times 1}$ (for single output)
- Role: Linear regression on the learned features

**Full network:**

$$f(\mathbf{x};\boldsymbol{\theta}) = f_2(f_1(\mathbf{x};\boldsymbol{\theta}_1);\boldsymbol{\theta}_2) \tag{78}$$

Total parameters: $(d \times k) + (k \times 1) = k(d+1)$

### 108.3 Loss Functions

**Common Loss Functions**

**Regression (MSE):**

$$\ell(y,\hat{y}) = (y-\hat{y})^2 \tag{79}$$

Measures squared deviation between prediction and target. Equivalently, $\mathcal{L} = \frac{1}{2}\|y-\hat{y}\|^2$ for vector outputs.

**Binary classification (binary cross-entropy / log loss):**

$$\ell(y,\hat{p}) = -y\log(\hat{p}) - (1-y)\log(1-\hat{p}) \tag{80}$$

where $\hat{p} \in (0,1)$ is the predicted probability (typically from sigmoid output). Suitable when $y \in \{0,1\}$.

**Multi-class classification (categorical cross-entropy):**

$$\ell(\mathbf{y},\hat{\mathbf{p}}) = -\sum_{k=1}^{K} y_k \log(\hat{p}_k) \tag{81}$$

where $\mathbf{y}$ is one-hot encoded and $\hat{\mathbf{p}}$ comes from softmax output.

## 109 Optimisers

While SGD is the foundation, several enhancements improve training speed and stability.

## 109.1 SGD with Momentum

Basic SGD can oscillate in ravines (directions with high curvature). **Momentum** accumulates gradients over time, smoothing updates:

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}, \quad \boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \mathbf{v}_t \tag{82}$$

where $\gamma \approx 0.9$ is the momentum coefficient. This accelerates convergence in consistent gradient directions and dampens oscillations.

## 109.2 Adam

Adam (Adaptive Moment Estimation) combines momentum with adaptive per-parameter learning rates.

**Adam Optimiser**

Adam maintains two moving averages:
**First moment (momentum):**

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{83}$$

Exponentially weighted average of gradients $g_t$. Acts like momentum, accumulating gradient direction.
**Second moment (adaptive scaling):**

$$s_t = \beta_2 s_{t-1} + (1 - \beta_2) g_t^2 \tag{84}$$

Exponentially weighted average of squared gradients. Tracks gradient magnitude per parameter.
**Bias correction** (important for early iterations):

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{s}_t = \frac{s_t}{1 - \beta_2^t} \tag{85}$$

**Parameter update:**

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \frac{\eta}{\sqrt{\hat{s}_t} + \epsilon} \hat{m}_t \tag{86}$$

The update is scaled inversely by the root mean square of recent gradients, giving each parameter an effective learning rate adapted to its gradient history.

**Adam Default Hyperparameters**

- $\beta_1 = 0.9$ (momentum decay)
- $\beta_2 = 0.999$ (scaling decay)
- $\epsilon = 10^{-6}$ or $10^{-8}$ (numerical stability)
- $\eta = 0.001$ or $0.003$ (learning rate)

Of these, the **learning rate** $\eta$ is the main hyperparameter to tune. The others rarely need adjustment.

### 109.3 BFGS

BFGS (Broyden-Fletcher-Goldfarb-Shanno) is a quasi-Newton method that approximates the Hessian matrix (second derivatives) to make more informed updates.

- **Advantages**: Can converge to exact solutions in fewer iterations; exploits curvature information
- **Disadvantages**: Memory grows as $O(p^2)$ where $p$ is the number of parameters; impractical for large networks

**Choosing an Optimiser**

**For small problems**: BFGS can rapidly converge to the exact solution, provided the problem size is manageable.
**For large-scale problems**: Adam (or SGD with momentum) efficiently handles large datasets and converges faster due to adaptive learning rates. This is the default choice for deep learning.

## 110 Universal Approximation

A fundamental theoretical result justifies the power of neural networks.

**Universal Approximation Theorem**

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, under mild assumptions on the activation function (non-constant, bounded, monotonically increasing—or non-polynomial for ReLU-type activations).
**Informal statement**: Given enough hidden units, a single hidden layer MLP can approximate any "reasonable" function as closely as desired.

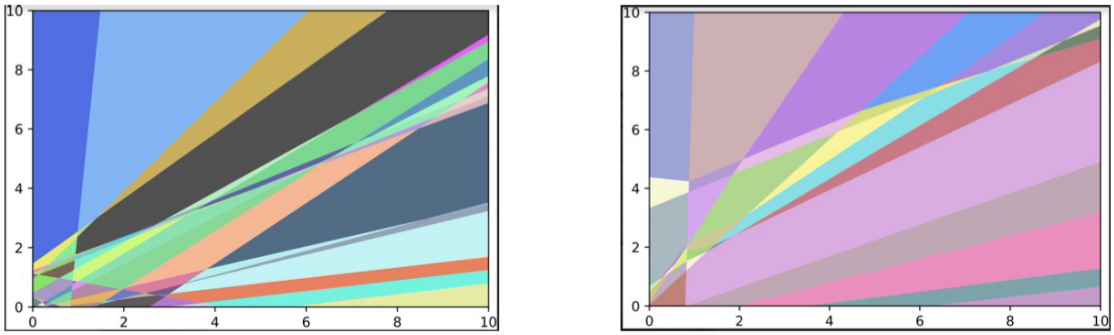



Figure 77: ReLU networks learn piecewise linear functions, partitioning input space into regions with different linear behaviours. More hidden units create finer partitions.

### 110.1 Theory vs Practice

**NB!**

[Depth vs Width] While one hidden layer is theoretically sufficient, in practice:

- The required width may be exponentially large in the input dimension
- Deep networks (multiple layers) often achieve the same accuracy with far fewer total parameters
- Deeper architectures naturally capture hierarchical/compositional structure in data

Why do deep networks work better empirically? Consider image recognition:

- Early layers learn low-level features (edges, textures)
- Middle layers combine these into parts (eyes, wheels)
- Later layers recognise objects (faces, cars)

This hierarchical structure matches the compositional nature of real-world data. A shallow network would need to learn all these patterns in one layer, requiring exponentially more units.

## 111 Neural Networks as Gaussian Processes

A remarkable theoretical connection exists between infinitely wide neural networks and Gaussian processes.

**Infinite-Width Limit**

As the number of hidden units approaches infinity, under appropriate scaling of weights:

1. The prior distribution over functions induced by random weight initialisation converges to a **Gaussian process**
2. The **Neural Tangent Kernel (NTK)** characterises this limit:

$$k(\mathbf{x}, \mathbf{y}) = \nabla_{\boldsymbol{\theta}} f(\mathbf{x}; \boldsymbol{\theta})^\top \mathbf{K} \nabla_{\boldsymbol{\theta}} f(\mathbf{y}; \boldsymbol{\theta}) \tag{87}$$

   where $\nabla_{\boldsymbol{\theta}} f(\mathbf{x}; \boldsymbol{\theta})$ is the gradient of the network output with respect to parameters
3. In this limit, the NTK remains constant during training
4. The network behaves like **kernel regression** with the NTK

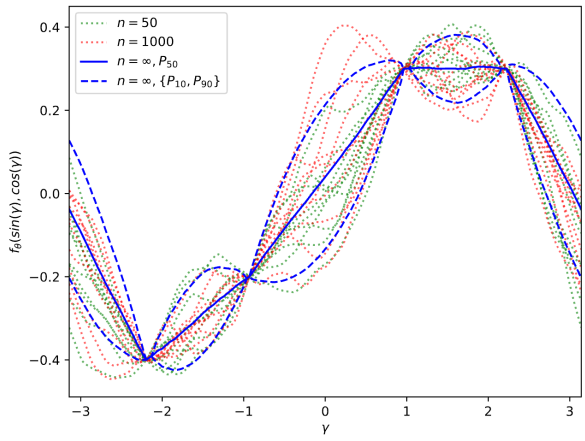



Figure 78: MLPs (red) as samples from a GP (blue) defined by the NTK. Different training trajectories (different minibatch sequences) yield different but related functions.

**Implications of the NTK**

- **Theoretical tool**: Provides rigorous analysis of neural network training dynamics
- **Architecture-dependent**: The NTK depends on depth, width, and activation functions
- **Bridges paradigms**: Connects parametric models (NNs) with non-parametric models (GPs)
- **Feature learning**: The MLP learns feature representations; in the infinite-width limit, this becomes regression on learned features

This framework provides powerful insights into how architecture choices affect learning, though practical networks are finite and exhibit more complex dynamics.

# 112 Vanishing and Exploding Gradients

As networks become deeper, gradient-based training faces fundamental challenges related to how gradients propagate through many layers.

## 112.1 The Problem

**Gradient Flow Through Depth**

By the chain rule, the gradient of the loss with respect to an early layer's activations is:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}_l} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}_L} \cdot \frac{\partial \mathbf{z}_L}{\partial \mathbf{z}_{L-1}} \cdots \frac{\partial \mathbf{z}_{l+1}}{\partial \mathbf{z}_l} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}_L} \prod_{i=l}^{L-1} \frac{\partial \mathbf{z}_{i+1}}{\partial \mathbf{z}_i} \tag{88}$$

If we assume (for simplicity) that the Jacobian between adjacent layers is approximately constant, $\frac{\partial \mathbf{z}_{i+1}}{\partial \mathbf{z}_i} \approx \mathbf{J}$, then:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}_l} \approx \mathbf{J}^{L-l} \frac{\partial \mathcal{L}}{\partial \mathbf{z}_L} \tag{89}$$

The behaviour of $\mathbf{J}^{L-l}$ as $L - l$ grows depends on the eigenvalues of $\mathbf{J}$:

**Eigenvalue Behaviour**

For a scalar analogy: $\lim_{k\to\infty} p^k = 0$ if $|p| < 1$, but $\lim_{k\to\infty} p^k = \infty$ if $|p| > 1$.
For matrices, the maximum eigenvalue $\lambda_{\max}$ controls the behaviour:

- $\lambda_{\max} < 1$: $\|\mathbf{J}^k\| \to 0$ exponentially (gradients vanish)
- $\lambda_{\max} > 1$: $\|\mathbf{J}^k\| \to \infty$ exponentially (gradients explode)
- $\lambda_{\max} = 1$: Gradients remain stable (the ideal but difficult to achieve)

**NB!**

[Consequences] **Vanishing gradients:**

- Earlier layers receive near-zero gradient signal
- Weights in early layers barely update
- Network fails to learn long-range dependencies

**Exploding gradients:**

- Gradient magnitudes grow uncontrollably
- Weight updates become enormous
- Training diverges; NaN values appear

## 112.2 Solution: Gradient Clipping (for Exploding Gradients)

**Gradient Clipping**

Limit gradient magnitudes before the update:

$$\mathbf{g} \leftarrow \min\left(1, \frac{c}{\|\mathbf{g}\|}\right)\mathbf{g} \tag{90}$$

where:

- $\mathbf{g}$ is the gradient vector
- $\|\mathbf{g}\|$ is its norm (typically L2)
- $c$ is the clipping threshold

If $\|\mathbf{g}\| > c$, the gradient is scaled down to have norm $c$. Otherwise, it is unchanged.

**Key properties:**

- **Direction preserved**: Clipping scales all components equally, maintaining the gradient direction
- **Adaptive learning rate**: Effectively reduces the learning rate when gradients are large
- **Robust training**: Prevents divergence while still allowing learning in the correct direction

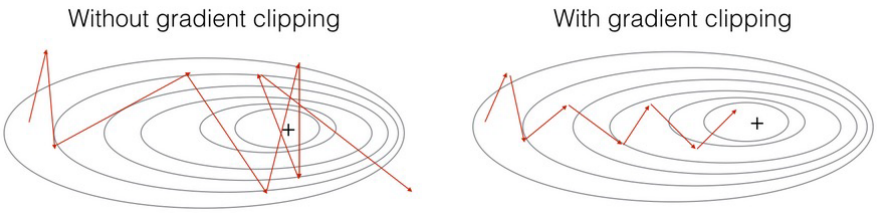



Figure 79: Gradient clipping constrains step size while preserving direction. The gradient magnitude is capped, but the direction of descent is maintained.

### 112.3 Solution: Non-Saturating Activations (for Vanishing Gradients)

The choice of activation function profoundly affects gradient flow.

**Activation Function Gradients**

**Sigmoid**:

$$\sigma(x) = \frac{1}{1+e^{-x}}, \quad \frac{d\sigma}{dx} = \sigma(x)(1-\sigma(x)) \tag{91}$$

The derivative is maximised at $\sigma(x) = 0.5$ (giving 0.25) and approaches 0 as $\sigma(x) \to 0$ or $\sigma(x) \to 1$. For large $|x|$, the sigmoid **saturates** and gradients vanish.

**ReLU**:

$$\text{ReLU}(x) = \max(0, x), \quad \frac{d\text{ReLU}}{dx} = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{92}$$

Gradient is 1 for all positive inputs—no saturation on the positive side. However, gradient is exactly 0 for negative inputs, leading to "dead neurons."

**Leaky ReLU**:

$$\text{LeakyReLU}(x) = \max(\alpha x, x), \quad \frac{d\text{LeakyReLU}}{dx} = \begin{cases} 1 & x > 0 \\ \alpha & x \leq 0 \end{cases} \tag{93}$$

where $\alpha \approx 0.01$. Gradient is never zero—prevents dead neurons while maintaining the benefits of ReLU.

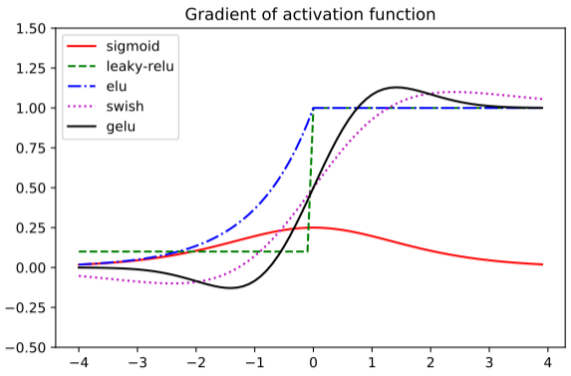



Figure 80: ReLU and Leaky ReLU maintain gradient flow better than sigmoid. Sigmoid saturates (gradient $\to 0$) for large inputs; ReLU maintains unit gradient for positive inputs.

**Summary: Activation Function Gradients**

- **Sigmoid**: Gradient vanishes when $z \approx 0$ or $z \approx 1$ (saturation at both extremes)
- **ReLU**: Gradient is 0 for negative inputs, but does not vanish on the positive side
- **Leaky ReLU**: Gradient is never zero ($\alpha$ for negative, 1 for positive)

**NB!**

[The Dead Neuron Problem] ReLU neurons that consistently receive negative inputs have zero gradient and never update—they are "dead." This can happen due to:

- Poor weight initialisation
- Large learning rates causing weights to overshoot
- Data distribution shifts during training

Leaky ReLU addresses this by ensuring a small but non-zero gradient for negative inputs, keeping all neurons "alive."

### 112.4 Choosing Between ReLU and Leaky ReLU

- **ReLU**: Sufficient for many applications, especially with careful initialisation and moderate depth. Simpler and slightly faster to compute.
- **Leaky ReLU**: Preferred when dead neurons are suspected, particularly in deeper networks or when training dynamics are unstable. The choice of $\alpha$ may require tuning.

## 113 Batch Normalisation

Batch normalisation, introduced by Ioffe and Szegedy in 2015, addresses the problem of **internal covariate shift**: the distribution of each layer's inputs changes during training as parameters in earlier layers update.
This shifting distribution can slow training (the network must constantly re-adapt) and requires careful initialisation and small learning rates.

### 113.1 The Idea

The core idea is to normalise layer activations to have zero mean and unit variance within each mini-batch, then allow the network to learn the optimal scale and shift.

**Batch Normalisation**

For a mini-batch of activations $\{x_1, \ldots, x_B\}$ at some layer:

**Step 1: Compute batch statistics**

$$\bar{x}_b = \frac{1}{B}\sum_{i=1}^{B} x_i, \quad \sigma_b^2 = \frac{1}{B}\sum_{i=1}^{B}(x_i - \bar{x}_b)^2 \tag{94}$$

**Step 2: Normalise**

$$\hat{x}_i = \frac{x_i - \bar{x}_b}{\sqrt{\sigma_b^2 + \epsilon}} \tag{95}$$

where $\epsilon$ is a small constant for numerical stability.

**Step 3: Scale and shift**

$$y_i = \gamma \hat{x}_i + \beta \tag{96}$$

where $\gamma$ (scale) and $\beta$ (shift) are **learnable parameters**.

The full transformation:

$$y = \frac{x - \bar{x}_b}{\sqrt{\sigma_b^2 + \epsilon}} \cdot \gamma + \beta \tag{97}$$

**Why learnable scale and shift?** After normalisation, all activations have mean 0 and variance 1. But this might not be optimal for the network's task. The learnable $\gamma$ and $\beta$ allow the network to recover the original distribution if beneficial, or find an even better one.

## 113.2 Behaviour at Test Time

During training, we compute batch statistics from each mini-batch. At test time, we may not have batches (or they may be very small), so we use **population statistics** estimated during training.

**Training vs Test Time**

**Training**: Use mini-batch mean $\bar{x}_b$ and variance $\sigma_b^2$ computed on the fly.

**Test time**: Use exponential moving averages accumulated during training:

$$\bar{x}_{\text{running}} \leftarrow \alpha \bar{x}_{\text{running}} + (1-\alpha)\bar{x}_b \tag{98}$$

$$\sigma^2_{\text{running}} \leftarrow \alpha \sigma^2_{\text{running}} + (1-\alpha)\sigma_b^2 \tag{99}$$

with typical $\alpha = 0.9$ or $0.99$.

## 113.3 Benefits of Batch Normalisation

1. **Faster convergence**: Reduces internal covariate shift, allowing higher learning rates
2. **Regularisation effect**: The noise from batch statistics (varying means/variances across batches) acts as a form of regularisation, sometimes reducing the need for dropout
3. **Improved gradient flow**: Normalising activations keeps them in a well-behaved range, mitigating vanishing/exploding gradients
4. **Reduced sensitivity to initialisation**: Less careful initialisation is needed when activations are normalised

# 114 Regularisation

Neural networks, with their many parameters, are prone to overfitting. Regularisation techniques encourage simpler models that generalise better.

## 114.1 Weight Decay (L2 Regularisation)

**Weight Decay**

Add a penalty on weight magnitudes to the loss:

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \frac{\lambda}{2}\|\mathbf{w}\|^2 \tag{100}$$

or equivalently:

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \frac{\lambda}{2}\mathbf{w}^\top\mathbf{w} \tag{101}$$

This penalises large weights, encouraging the network to use smaller, more distributed weights.

**Intuition**: Large weights mean the network is placing heavy reliance on specific features or connections. Penalising large weights encourages the network to spread influence across many features, leading to more robust predictions.
**Implementation**: Weight decay is often built directly into optimisers. In SGD, the update becomes:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta(\nabla_{\mathbf{w}}\mathcal{L} + \lambda\mathbf{w}) = (1 - \eta\lambda)\mathbf{w} - \eta\nabla_{\mathbf{w}}\mathcal{L} \tag{102}$$

The term $(1 - \eta\lambda)$ shrinks weights toward zero at each update—hence "weight decay."

## 114.2 Dropout

**Dropout**

During training, randomly "drop out" (set to zero) a fraction $p$ of activations at each layer:

$$\tilde{h}_i = \begin{cases} 0 & \text{with probability } p \\ h_i/(1-p) & \text{with probability } 1-p \end{cases} \tag{103}$$

The scaling by $1/(1-p)$ ensures the expected value of activations remains unchanged.
**At test time**: Use all neurons (no dropout), with activations as-is (since we scaled during training).

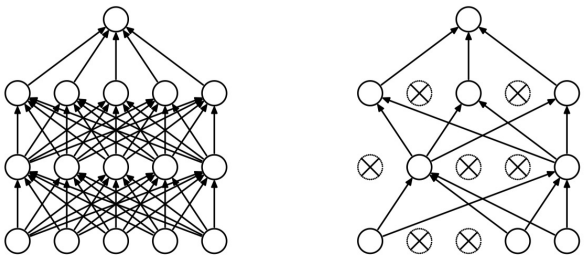


Figure 81: Dropout randomly removes connections during training, creating a different "thinned" network at each forward pass.

**Why dropout works**:

1. **Prevents co-adaptation**: Neurons cannot rely on specific other neurons always being present; they must learn more robust features
2. **Implicit ensemble**: Training with dropout is like training an ensemble of $2^n$ different networks (where $n$ is the number of neurons that can be dropped), then averaging their predictions at test time
3. **Redundant representations**: Forces the network to learn distributed representations where information is spread across many neurons

**Combining Regularisation Techniques**

Weight decay and dropout are complementary:

- **Weight decay**: Constrains weight magnitudes, favouring simpler linear relationships
- **Dropout**: Prevents co-adaptation, encouraging distributed representations

Both can be used together. Typical dropout rates are $p = 0.2$ to $0.5$, with higher rates for larger layers. Weight decay coefficients ($\lambda$) are typically $10^{-4}$ to $10^{-2}$.

## 115 Summary

**Key Concepts from Week 10**

1. **Perceptrons**: Linear classifiers using hard thresholds; limited to linearly separable data (cannot solve XOR)
2. **Multi-layer networks**: Learn feature representations through composed layers; overcome perceptron limitations
3. **Non-linearity is essential**: Without it, stacking layers provides no benefit—the network collapses to a single linear transformation
4. **Activation functions**: ReLU is the default; sigmoid/tanh saturate and cause vanishing gradients; Leaky ReLU prevents dead neurons
5. **Backpropagation**: Efficient gradient computation via chain rule; forward pass stores activations, backward pass propagates gradients
6. **Automatic differentiation**: Modern frameworks (PyTorch, TensorFlow, JAX) compute gradients automatically by tracking operations
7. **Optimisers**: Adam (adaptive learning rates) for large problems; BFGS (second-order) for small problems; SGD with momentum as a baseline
8. **Universal approximation**: One hidden layer can approximate any continuous function, but deeper networks are more efficient in practice
9. **NTK connection**: Infinitely wide networks converge to Gaussian processes; provides theoretical analysis tools
10. **Gradient problems**: Vanishing (use non-saturating activations like ReLU); Exploding (use gradient clipping)
11. **Batch normalisation**: Normalises layer inputs; enables higher learning rates; provides regularisation
12. **Regularisation**: Weight decay (L2 penalty) and dropout (random deactivation) prevent overfitting; often used together

# 116 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**: Enable flexible, learnable weighting of inputs

The unifying theme is **composability**: building complex representations from simple, reusable components tailored to data structure.

# 117 Neural Network Design Recap

## 117.1 Architecture Components

**Network Design Choices**

**Inputs:**

- Input layer dimensionality matches feature count
- Example: $28 \times 28$ image $\to$ 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

### 117.2 Loss Functions

**Standard Loss Functions**

**Regression—Mean Squared Error:**

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

**Binary Classification—Cross-Entropy (Log Loss):**

$$\mathcal{L}_{\text{CE}} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$$

### 117.3 Optimisers

**Gradient-Based Optimisation**

**Stochastic Gradient Descent (SGD):**

$$w_{t+1} = w_t - \eta \nabla \mathcal{L}(w_t)$$

where $\eta$ is the learning rate.
**Adam** combines momentum with adaptive learning rates:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \\ w_{t+1} &= w_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \end{aligned}$$

Adam is generally the default choice for deep learning. Other common optimisers include BFGS (second-order, but expensive for large models) and AdaGrad/RMSProp (which Adam builds upon).

**Interactive exploration:** TensorFlow Playground (`playground.tensorflow.org`) provides an excellent interactive environment for building intuition about how network architecture, activation functions, and optimisation interact.

## 118 Vanishing and Exploding Gradients

Gradient-based training propagates error signals backward through layers via the chain rule. In deep networks, this can become unstable.

## 118.1 The Problem

**Gradient Flow in Deep Networks**

For a network with $L$ layers, backpropagation computes derivatives using the chain rule. Let $z_l$ denote the pre-activation at layer $l$ (i.e., the weighted sum before applying the activation function). The gradient of the loss with respect to $z_l$ is:

$$\frac{\partial \mathcal{L}}{\partial z_l} = \frac{\partial \mathcal{L}}{\partial z_L} \cdot \frac{\partial z_L}{\partial z_{L-1}} \cdot \frac{\partial z_{L-1}}{\partial z_{L-2}} \cdots \frac{\partial z_{l+1}}{\partial z_l}$$

This can be written compactly as:

$$\frac{\partial \mathcal{L}}{\partial z_l} = \frac{\partial \mathcal{L}}{\partial z_L} \cdot \prod_{k=l}^{L-1} \frac{\partial z_{k+1}}{\partial z_k}$$

**Key insight:** The gradient at layer $l$ depends on a *product* of $L - l$ terms. If these terms are consistently greater than or less than 1, the product grows or shrinks exponentially with depth.

**Gradient Behaviour with Depth**

If layer-wise derivatives are approximately constant $\frac{\partial z_{k+1}}{\partial z_k} \approx J$, then:

$$\frac{\partial \mathcal{L}}{\partial z_l} \approx J^{L-l} \cdot \frac{\partial \mathcal{L}}{\partial z_L}$$

For a matrix $J$, the behaviour depends on its eigenvalues $\lambda$:

- $|\lambda| < 1$: $\lim_{L\to\infty} J^{L-l} = 0$ (vanishing gradients)
- $|\lambda| > 1$: $\lim_{L\to\infty} J^{L-l} = \infty$ (exploding gradients)

**Practical consequences:**

- **Vanishing gradients**: Early layers receive negligible updates; the network stops learning useful representations in lower layers.
- **Exploding gradients**: Weight updates become erratic; training diverges with NaN losses.

## 118.2 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**.

**Direction preservation:** The key insight is that clipping scales all gradient components uniformly, so the optimisation continues moving in the correct direction—just with a bounded step size. This prevents wild jumps in parameter space while maintaining the correct "heading".

**Interpretation as adaptive learning rate:** Gradient clipping can be viewed as dynamically reducing the effective learning rate when gradients are large. When $\|g\| > c$:

$$g' = \frac{c}{\|g\|} \cdot g \implies \text{effective learning rate} = \eta \cdot \frac{c}{\|g\|}$$

Large gradients automatically trigger smaller steps, stabilising training.

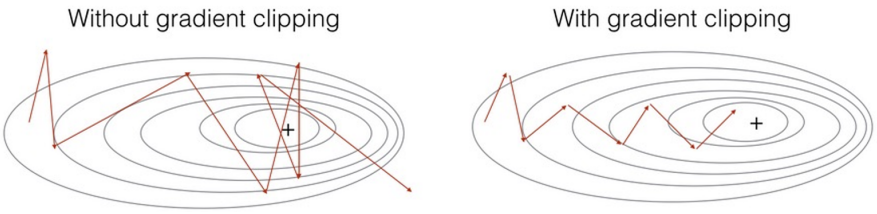



Figure 82: Gradient clipping constrains update magnitude while preserving direction, preventing divergence in optimisation. Without clipping, large gradients can cause the optimisation to "overshoot" and diverge.

## 118.3 Vanishing Gradients and Activation Functions

The choice of activation function critically affects gradient flow. Saturating activations (like sigmoid) compress their output range, which also compresses gradients.

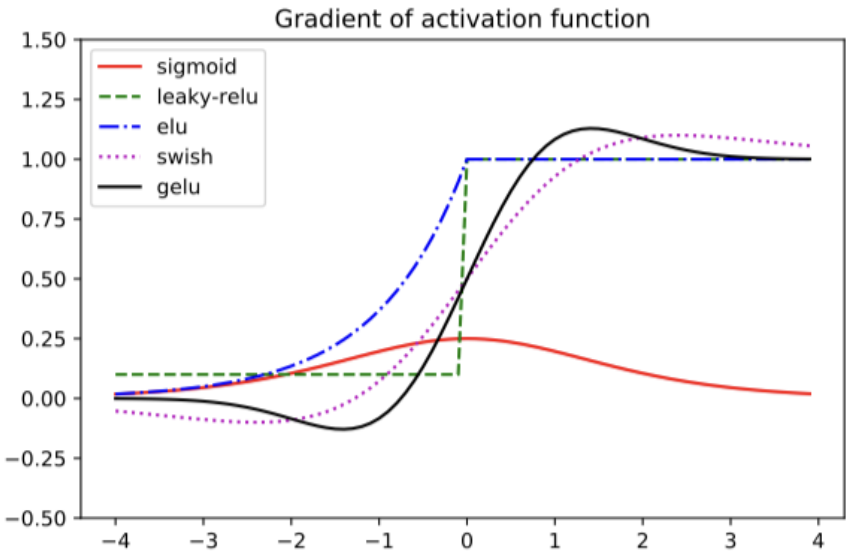



Figure 83: Activation functions and their derivatives. Sigmoid saturates for large inputs, causing vanishing gradients. ReLU maintains gradient of 1 for positive inputs.

**Activation Function Gradients**

**Sigmoid:** $\sigma(x) = \frac{1}{1+e^{-x}}$
The derivative is:

$$\frac{d\sigma}{dx} = \sigma(x)(1 - \sigma(x))$$

Since $\sigma(x) \in (0, 1)$, the maximum derivative is 0.25 (when $\sigma(x) = 0.5$). For large $|x|$, $\sigma(x) \to 0$ or $\sigma(x) \to 1$, and the gradient approaches zero (saturation).
**Intuition:** In deep networks, if inputs become large at any layer, the sigmoid saturates and gradients vanish. This happens repeatedly through the network, compounding the problem.
**ReLU:** $\text{ReLU}(x) = \max(0, x)$

$$\frac{d\text{ReLU}}{dx} = \mathbb{I}(x > 0) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$

Gradient is exactly 1 for positive inputs (no saturation), but exactly 0 for negative inputs. The zero gradient for negative inputs can cause "dead neurons" that never activate.
**Leaky ReLU:** $\text{LeakyReLU}(x) = \max(\alpha x, x)$ where typically $\alpha = 0.01$

$$\frac{d\text{LeakyReLU}}{dx} = \begin{cases} 1 & x > 0 \\ \alpha & x \leq 0 \end{cases}$$

Non-zero gradient everywhere—addresses both vanishing gradients (for $x > 0$) and dead neurons (for $x \leq 0$).

**Activation Function Summary**

- **Sigmoid**: Vanishes for large $|x|$; avoid in hidden layers of deep networks. May still be useful when smoothness is specifically required.
- **ReLU**: No saturation for $x > 0$; risk of "dead neurons" when $x < 0$ permanently. Most commonly used due to simplicity and effectiveness.
- **Leaky ReLU**: Best of both worlds—non-saturating and no dead neurons. Use when dead neurons are a concern.

In practice, combining different activation functions can improve performance and stability.

## 118.4 Batch Normalisation

Batch normalisation stabilises training by normalising layer activations, preventing them from drifting to saturating regions.

## Batch Normalisation

For a mini-batch of activations $\{x_1, \ldots, x_m\}$ at a given layer:

**Step 1: Compute batch statistics**

$$\bar{x}_{\text{batch}} = \frac{1}{m}\sum_{i=1}^{m} x_i, \qquad \sigma^2_{\text{batch}} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \bar{x}_{\text{batch}})^2$$

**Step 2: Normalise**

$$\hat{x}_i = \frac{x_i - \bar{x}_{\text{batch}}}{\sqrt{\sigma^2_{\text{batch}} + \epsilon}}$$

**Step 3: Scale and shift (learnable parameters)**

$$y_i = \gamma \hat{x}_i + \beta$$

where:

- $\bar{x}_{\text{batch}}$, $\sigma^2_{\text{batch}}$ = batch mean and variance
- $\gamma$, $\beta$ = learned scale and shift parameters
- $\epsilon$ = small constant for numerical stability (typically $10^{-5}$)

**Why learnable $\gamma$ and $\beta$?** The normalisation step forces activations to have zero mean and unit variance. But this might not be optimal—perhaps the network would benefit from activations with different statistics. The learnable parameters $\gamma$ and $\beta$ allow the network to "undo" the normalisation if beneficial, recovering the original distribution when $\gamma = \sigma_{\text{batch}}$ and $\beta = \bar{x}_{\text{batch}}$.

**At test time:** We cannot compute batch statistics from a single example. Instead, use exponential moving averages of training statistics:

$$\bar{x}_{\text{test}} = \text{EMA of } \bar{x}_{\text{batch}}, \qquad \sigma^2_{\text{test}} = \text{EMA of } \sigma^2_{\text{batch}}$$

These moving averages are updated throughout training and then fixed at test time.

## Benefits of Batch Normalisation

- **Reduces internal covariate shift**: Activation distributions stay stable across training, making optimisation easier
- **Allows higher learning rates**: More stable gradients permit more aggressive optimisation
- **Acts as implicit regularisation**: The noise from batch statistics adds a regularising effect
- **Prevents saturation**: Keeps activations in the "active" region of activation functions (neither too large nor too small)

## NB!

Batch normalisation behaviour differs between training and inference. Always ensure your model is in the correct mode (`model.train()` vs `model.eval()` in PyTorch). Using training-mode batch norm at inference time produces inconsistent results because batch statistics vary with each input.

## 118.5 Regularisation

### 118.5.1 Weight Decay (L2 Regularisation)

**Weight Decay**

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \frac{\lambda}{2}\|w\|^2$$

Equivalently (collecting all weights into a vector):

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \frac{\lambda}{2} w^\top w$$

where $\lambda$ is the regularisation coefficient controlling the strength of the penalty.

**Interpretation:** Weight decay penalises large weights, encouraging simpler models. This is directly analogous to ridge regression applied to all network parameters. Large weights allow the network to fit complex, potentially spurious patterns; constraining weight magnitude encourages smoother functions that generalise better.
**Implementation:** Weight decay is typically implemented directly *in the optimiser* rather than added to the loss function. In PyTorch, this is the `weight_decay` parameter in optimiser constructors. This is more computationally efficient and avoids numerical issues.

### 118.5.2 Dropout

**Dropout**

During training, randomly set each neuron's output to zero with probability $p$:

$$\tilde{h}_i = \frac{1}{1-p} \cdot h_i \cdot \text{Bernoulli}(1-p)$$

The factor $\frac{1}{1-p}$ ("inverted dropout") ensures expected values match between training and test time.
At test time, use all neurons (no dropout) without any scaling adjustment.

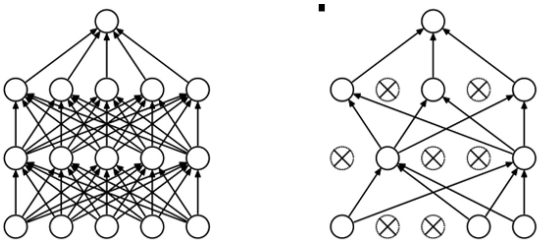


Figure 84: Dropout randomly removes neurons during training, forcing redundant representations. Each training iteration uses a different random subnetwork.

**Intuition:** Dropout prevents *co-adaptation* of neurons. Without dropout, neurons can learn to rely on specific other neurons being present. Dropout forces each neuron to be useful on its own, building redundancy into the network.
**Ensemble interpretation:** Dropout can be viewed as implicitly training an ensemble of $2^n$ subnetworks (where $n$ is the number of neurons), each corresponding to a different dropout mask. At test time, using all neurons with appropriately scaled weights approximates averaging predictions from this exponential ensemble.

**Uncertainty estimation:** Applying dropout at test time (rather than turning it off) and averaging multiple forward passes provides a form of uncertainty quantification—predictions that vary significantly under different dropout masks indicate model uncertainty.

**Regularisation Summary**

- **Weight decay**: Encourages small weights; simpler, smoother functions
- **Dropout**: Forces redundancy; prevents co-adaptation of neurons

Both techniques reduce overfitting and build robustness into the learned representations.

# 119 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

The core operation is **convolution**: sliding a small filter across the input to extract local features.

## 119.1 Why Image Data is Challenging

**High dimensionality:** A $256 \times 256$ RGB image has $256 \times 256 \times 3 = 196{,}608$ features—and they are continuous. A fully connected layer from this input to even 1000 hidden units would require nearly 200 million parameters.
**Local structure matters:** Unlike tabular data where feature order is often arbitrary, images have meaningful spatial relationships. Neighbouring pixels are correlated; edges and textures emerge from local patterns. A pixel's meaning depends on its neighbours.
**Comparison with other data types:**

- In ridge regression or random forests, feature order is irrelevant—we can permute columns without changing the model
- In images, spatial relationships are fundamental to meaning; permuting pixels destroys the image

**Translation invariance:** A face should be recognised regardless of its position in the image. Fully connected networks treat each pixel position independently, requiring the network to learn the same pattern separately for each possible location—massively inefficient.
**Continuous features:** Pixel intensities are continuous (typically 0–255 or 0–1), unlike categorical features common in tabular data. This continuity requires different treatment than one-hot encoded categories.

## 119.2 The Convolution Operation

The key insight: instead of processing the entire image at once, slide a small filter across the image to extract local features.

**Convolution Definition**

**Continuous (1D):**

$$(f * g)(z) = \int_{\mathbb{R}} f(u)\, g(z-u)\, du$$

**Discrete 1D** (filter $w$ with $k$ elements applied to input $x$):

$$(w * x)_i = \sum_{j=1}^{k} w_j \cdot x_{i+j-1}$$

This computes a *locally weighted sum*: each output position is a weighted combination of nearby input values, where the weights are the filter values.

**Discrete 2D** (filter $W$ applied to input $X$):

$$(W * X)_{i,j} = \sum_{m} \sum_{n} W_{m,n} \cdot X_{i+m,j+n}$$

The filter slides across the input, computing a weighted sum at each position.

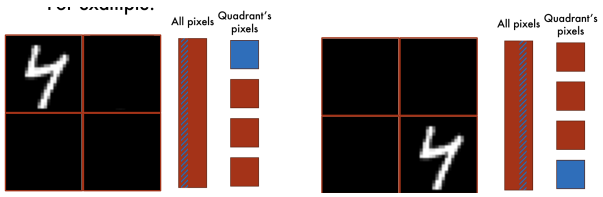



Figure 85: A convolution filter sliding across an image, computing local weighted sums. The filter "looks at" one small region at a time.

**The problem with simple partitioning:** What if we simply divided the image into non-overlapping regions? Objects that span region boundaries would be split, making them hard to recognise.

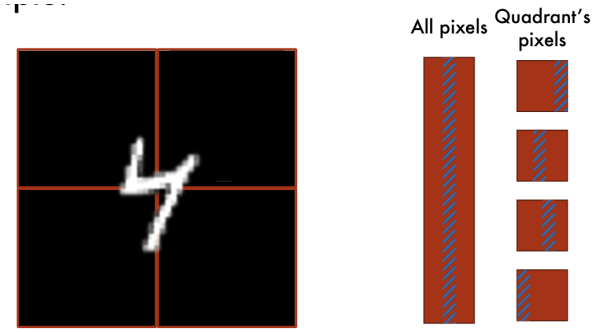



Figure 86: When objects span multiple regions, convolutions capture features that simple partitioning would miss. The sliding window ensures every local pattern is detected regardless of position.

**Solution:** Convolutions use 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.

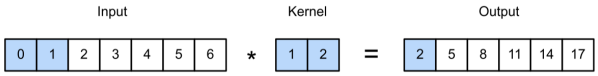



Figure 87: 1D convolution: the filter weights determine which local patterns are detected. Different weights detect different patterns (e.g., edges, gradients, flat regions).

$$[W * X](i,j) = \sum_{u=0}^{H-1} \sum_{v=0}^{W-1} w_{u,v} x_{i+u,j+v}$$

Figure 88: 2D convolution for image processing, extending the same principle to spatial data. The filter has both width and height.

**Learned filters:** The network *learns* the filter weights during training—we do not hand-design them. Different filters detect different features: edges, textures, shapes. Early convolutional layers typically learn edge detectors; deeper layers learn more complex patterns.

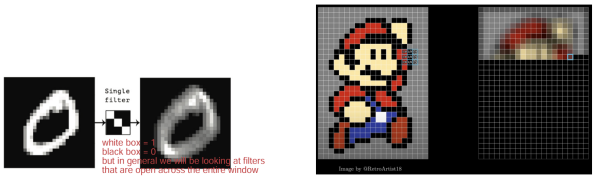



Figure 89: Learned filters detect interpretable features such as horizontal and vertical edges, colour gradients, and textures.

## 119.3 Convolution as Sparse Matrix Multiplication

**Matrix View of Convolution**

Convolution can be expressed as matrix multiplication $y = Cx$ where $C$ is a sparse, structured matrix:

$$C = \begin{bmatrix} w_1 & w_2 & 0 & 0 & \cdots \\ 0 & w_1 & w_2 & 0 & \cdots \\ 0 & 0 & w_1 & w_2 & \cdots \\ \vdots & & & & \ddots \end{bmatrix}$$

**Two key properties:**

1. **Weight sharing**: The same weights $(w_1, w_2, \ldots)$ appear in every row—the filter is reused across all positions
2. **Sparsity**: Most entries are zero—each output depends only on local inputs

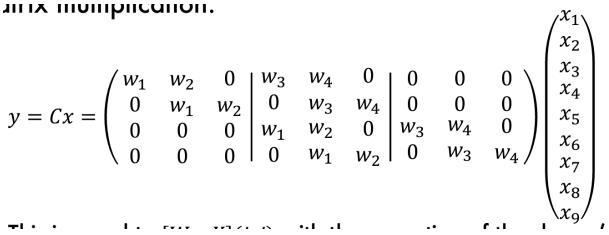



Figure 90: Convolution as sparse matrix multiplication: zeros correspond to pixels outside the receptive field. We only "pay attention" to features within the filter's window.

**Parameter efficiency:** A fully connected layer from a $100 \times 100$ image to a $100 \times 100$ output requires $10^8$ parameters. A $3 \times 3$ convolution achieving the same output shape requires only 9 parameters (plus one bias). This dramatic reduction encodes the inductive bias that local patterns matter more than distant correlations.

## 119.4 Padding and Strides

### 119.4.1 Padding

**Padding Types**

**Valid convolution:** No padding. Output shrinks because the filter cannot be centred on edge pixels—it would extend beyond the image boundary.
**Same convolution:** Zero-pad the input so output dimensions match input dimensions. Padding width is typically $\lfloor k/2 \rfloor$ for a $k \times k$ filter.

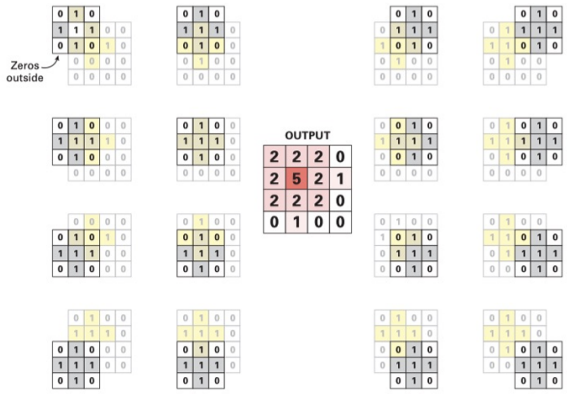



Figure 91: Zero-padding allows the filter to process edge regions, preserving spatial dimensions. The added zeros (shown in grey) extend the input so the filter can be centred on boundary pixels.

**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.

### 119.4.2 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**

- **Stride 1**: Filter moves one pixel per step (default). Adjacent outputs overlap by $(k-1)$ pixels for a $k \times k$ filter.
- **Stride 2**: Filter skips one pixel; output spatial dimensions halved.
- **Stride $s$**: Output dimensions reduced by factor of $s$.

For input dimension $n$, filter size $k$, padding $p$, and stride $s$:

$$\text{output dimension} = \left\lfloor \frac{n + 2p - k}{s} \right\rfloor + 1$$

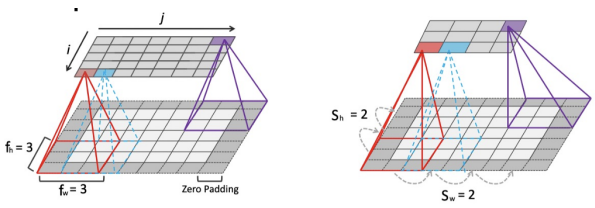



Figure 92: Comparison of stride 1 (top) vs stride 2 (bottom). Larger strides downsample the feature map, reducing computation and introducing some translation invariance.

**Intuition:** With stride 1, adjacent outputs share about 90% of their receptive field (for a $3 \times 3$ filter). This redundancy is computationally expensive. Larger strides reduce this overlap, producing a more compact representation at the cost of spatial resolution.

## 119.5 Pooling

Pooling provides translation invariance by summarising local regions.

**Pooling Operations**

For a $2 \times 2$ region with values $(a, b, c, d)$:

- **Max pooling**: output $= \max(a, b, c, d)$
- **Average pooling**: output $= \frac{1}{4}(a + b + c + d)$

Max pooling is more common; it selects the strongest activation in each region, emphasising the presence of features rather than their exact location.

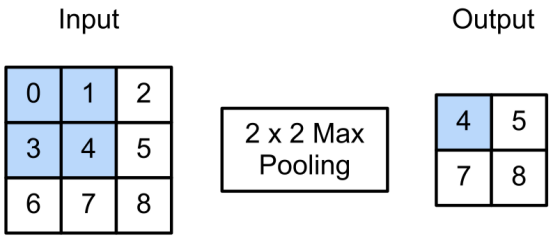



Figure 93: Max pooling selects the largest value in each region, providing translation robustness. Small shifts in the input often produce the same pooled output.

**Translation invariance:** Pooling makes the network robust to small translations. If an edge shifts by one pixel, the max within a $2 \times 2$ region is likely unchanged. This helps the network generalise across object positions.
**Deliberate information loss:** Pooling intentionally discards spatial precision. This is a feature, not a bug—we want the network to care about *what* features are present, not 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 means only the "winning" neurons get updated, which can sometimes cause training instabilities.

## 119.6 CNN Architecture

A typical CNN alternates convolutional and pooling layers, building a hierarchy of increasingly abstract features.

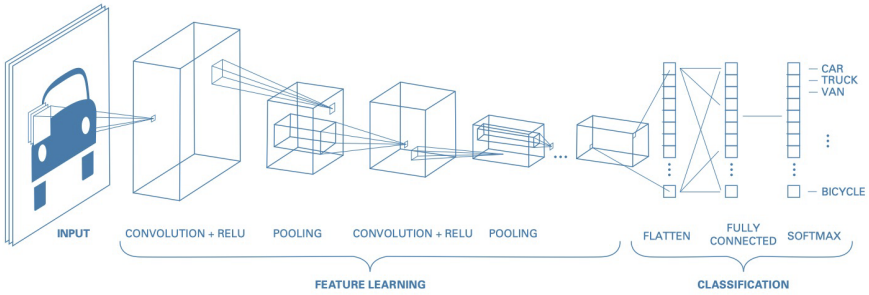



Figure 94: CNN architecture: early layers detect edges, middle layers detect parts (eyes, wheels), deep layers detect objects (faces, cars). This hierarchical feature learning is a key strength of CNNs.

**Standard CNN Structure**

1. **Convolutional layers**: Extract local features with learned filters. Multiple filters per layer capture different patterns.
2. **Activation functions**: Apply non-linearity (typically ReLU) after each convolution.
3. **Pooling layers**: Downsample and provide translation invariance. Typically $2 \times 2$ max pooling with stride 2.
4. **Repeat**: Stack conv/pool blocks to build hierarchical representations. Deeper = more abstract features.
5. **Fully connected layers**: Flatten final feature maps and map to outputs (classification logits or regression values).

**Hierarchical feature learning:**

- **Early layers**: Low-level features (edges, textures, colour gradients)
- **Middle layers**: Mid-level features (parts, shapes, simple objects)
- **Deep layers**: High-level features (objects, concepts, scenes)

This progression from specific local patterns to generic high-level concepts mirrors how biological visual systems work and is key to CNNs' success.

**When CNNs Work Well**

CNNs excel when:

- **Local structure matters**: Neighbouring elements are correlated (edges form from adjacent pixels)
- **Translation invariance is desired**: Pattern location is less important than pattern presence
- **Hierarchy exists**: Complex patterns compose from simpler ones (faces from eyes, noses, mouths)

**Example applications**: Images, spectrograms, genomic sequences, 2D/3D medical imaging, satellite imagery.

**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. The hierarchical nature means global context only emerges at the very top of the network.
Use RNNs, LSTMs, or Transformers for tasks where:

- Global sequence order is critical
- Long-distance relationships matter (e.g., coreference in text)
- Context from distant positions affects interpretation

## 120 Recurrent Neural Networks

**RNNs: Key Idea**

RNNs maintain **hidden state** that evolves over time, enabling:

- Variable-length input/output sequences (unlike fixed-size inputs for MLPs/CNNs)
- Implicit dependence on all previous inputs via the hidden state
- Memory of past context for making current predictions

## 120.1 Architecture

**RNN Formulation**

At each time step $t$:

- Receive input $x_t$ (e.g., a word embedding, a time series value, a one-hot character)
- Receive previous hidden state $h_{t-1}$
- Compute new hidden state and output:

$$(h_t, \hat{y}_t) = f(x_t, h_{t-1}; \theta)$$

The function $f$ is typically:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$
$$\hat{y}_t = W_{hy}h_t + b_y$$

**Initialisation:** $h_0 = \mathbf{0}$ (or learned initial state)
**Training:** Backpropagation through time (BPTT)—unroll the network across time steps and apply standard backpropagation.

**The key insight:** The hidden state $h_t$ acts as a "memory" that summarises all previous inputs. This allows the network to use past context when processing the current input, without explicitly storing the entire history.

## 120.2 Properties and Capabilities

**Variable input length:** RNNs process one element at a time, accumulating information in the hidden state. No fixed input dimension required—ideal for text (varying sentence lengths), time series (varying durations), and other sequential data.
**Implicit dependence on history:** Through the hidden state, each output implicitly depends on all previous inputs. The network can (in principle) remember relevant information from arbitrarily far in the past.
**Theoretical expressiveness:** With sufficient hidden state capacity, RNNs are Turing-complete—they can represent any computable function of the input sequence. This makes them theoretically very powerful.

**NB!**

**Practical limitations:**

- **Limited memory capacity**: The hidden state has finite dimension; information degrades as sequences get longer
- **Vanishing/exploding gradients**: Gradients must flow through many time steps, causing the same issues as deep networks
- **Sequential processing**: Each step depends on the previous, preventing parallelisation across time
- **Difficulty learning long-range dependencies**: Important information from early in a sequence may not survive to influence later predictions

These limitations motivated LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit) architectures, which use gating mechanisms to better preserve information. Ultimately, Transformers largely replaced RNNs for many sequence tasks by enabling parallel processing and direct attention across all positions.

## 121 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 that adapt to each input
- Direct access to all positions (no sequential bottleneck like RNNs)
- Interpretable importance weights showing what the model "focuses on"

Attention is the building block of modern deep learning, particularly Transformers.

**Attention Is All You Need**

**Ashish Vaswani*** Google Brain avaswani@google.com
**Noam Shazeer*** Google Brain noam@google.com
**Niki Parmar*** Google Research nikip@google.com
**Jakob Uszkoreit*** Google Research usz@google.com
**Llion Jones*** Google Research llion@google.com
**Aidan N. Gomez*** † University of Toronto aidan@cs.toronto.edu
**Łukasz Kaiser*** Google Brain lukaszkaiser@google.com
**Illia Polosukhin*** ‡ illia.polosukhin@gmail.com

**Abstract**

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.0 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature.

Figure 95: Attention mechanism: queries attend to keys, retrieving weighted combinations of values. The attention weights determine how much each value contributes to the output.

## 121.1 General Attention

**Attention Definition**

Given:

- **Query** $q$: what we're looking for (a set of features describing our "question")
- **Keys** $k_1, \ldots, k_n$: features describing each available piece of data
- **Values** $v_1, \ldots, v_n$: the actual content to retrieve (could be labels, embeddings, etc.)
- **Similarity function** $\phi(\cdot, \cdot)$: measures how well a query matches each key

Attention output:

$$\text{Attn}(q, \{k_i, v_i\}) = \sum_{i=1}^{n} \underbrace{\frac{\phi(q, k_i)}{\sum_{j=1}^{n} \phi(q, k_j)}}_{\text{attention weight } \alpha_i} \cdot v_i$$

The weights $\alpha_i = \frac{\phi(q,k_i)}{\sum_j \phi(q,k_j)}$ sum to 1, forming a probability distribution over values.

**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 of all values, with weights based on similarity to the query. Similar keys contribute more to the output.
**Interpretation as kernel-weighted average:** The similarity function $\phi$ acts like a kernel, measuring proximity between query and keys. The output is then a kernel-smoothed average of the values—similar to Nadaraya-Watson kernel regression, but with learned representations.
**Example: Machine translation**
Consider translating "The cat sat on the mat" to French. When generating the French word for "cat":

- The **query** represents the decoder's current state ("what word should come next?")

- The **keys** represent each English word's encoding
- The **values** are the English word embeddings
- The attention mechanism "looks at" the most relevant English words (mainly "cat") to inform the translation

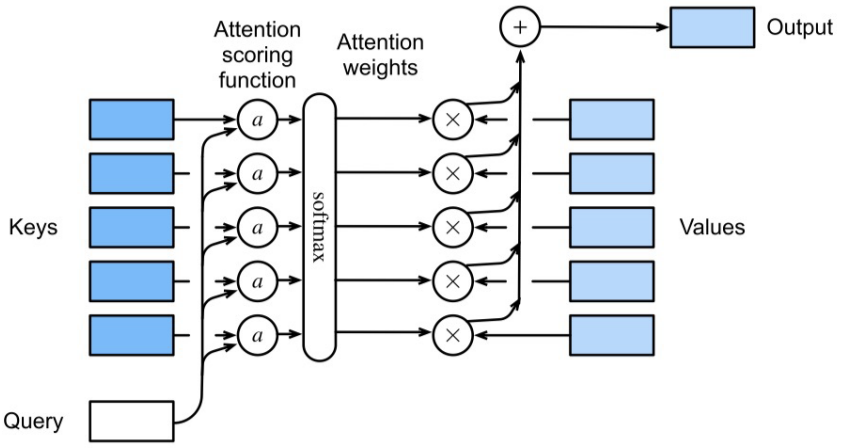



Figure 96: Attention weights visualised: each query attends differently to the available keys. Brighter colours indicate higher attention weights.

## 121.2 Scaled Dot-Product Attention

The most common attention implementation uses dot products for similarity, with a crucial scaling factor.

**Scaled Dot-Product Attention**

For query matrix $Q \in \mathbb{R}^{m \times d}$, key matrix $K \in \mathbb{R}^{n \times d}$, value matrix $V \in \mathbb{R}^{n \times p}$:

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

**Step-by-step:**

1. **Compute similarities**: $QK^\top \in \mathbb{R}^{m \times n}$ gives dot products between all query-key pairs
2. **Scale**: Divide by $\sqrt{d}$ to stabilise magnitudes
3. **Normalise**: Apply softmax row-wise to get attention weights (each row sums to 1)
4. **Aggregate**: Multiply by $V$ to get weighted combinations of values

Output shape: $\mathbb{R}^{m \times p}$ (one $p$-dimensional output per query).

### 121.2.1 Why Scale by $\sqrt{d}$?

This scaling is crucial for training stability.

**Scaling Justification**

If query and key elements are approximately standard normal (mean 0, variance 1), their dot product is:

$$q^\top k = \sum_{i=1}^{d} q_i k_i$$

Each term $q_i k_i$ has mean 0 and variance 1. By independence, the sum has:

$$\mathrm{Var}(q^\top k) = d$$

For large $d$, dot products can be very large in magnitude (order $\sqrt{d}$). After softmax, large magnitudes push the output towards one-hot vectors, where gradients become very small (the softmax "saturates").
Dividing by $\sqrt{d}$ normalises the variance to 1, keeping dot products in a stable range regardless of dimensionality.

### 121.2.2 Row-wise Softmax

**Softmax Operation**

For matrix $X \in \mathbb{R}^{m \times n}$:

$$\mathrm{softmax}(X)_{ij} = \frac{\exp(x_{ij})}{\sum_{k=1}^{n} \exp(x_{ik})}$$

Each row is normalised independently to sum to 1. This converts raw similarity scores into a probability distribution over keys for each query.
Properties:

- Output entries are positive and sum to 1 per row
- Higher input values get exponentially higher weights
- Preserves ordering: if $x_{ij} > x_{ik}$, then $\mathrm{softmax}(X)_{ij} > \mathrm{softmax}(X)_{ik}$

## 121.3 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 $n$ elements, each $d$-dimensional), compute queries, keys, and values via learned projections:

$$\begin{aligned} Q &= XW_Q \quad (W_Q \in \mathbb{R}^{d \times d_k}) \\ K &= XW_K \quad (W_K \in \mathbb{R}^{d \times d_k}) \\ V &= XW_V \quad (W_V \in \mathbb{R}^{d \times d_v}) \end{aligned}$$

Then apply scaled dot-product attention:

$$\text{SelfAttn}(X) = \text{softmax}\left(\frac{XW_Q(XW_K)^\top}{\sqrt{d_k}}\right) XW_V$$

Equivalently:

$$\text{SelfAttn}(X) = \text{softmax}\left(\frac{XW_Q W_K^\top X^\top}{\sqrt{d_k}}\right) XW_V$$

Output shape: $\mathbb{R}^{n \times d_v}$ (one output per input position).

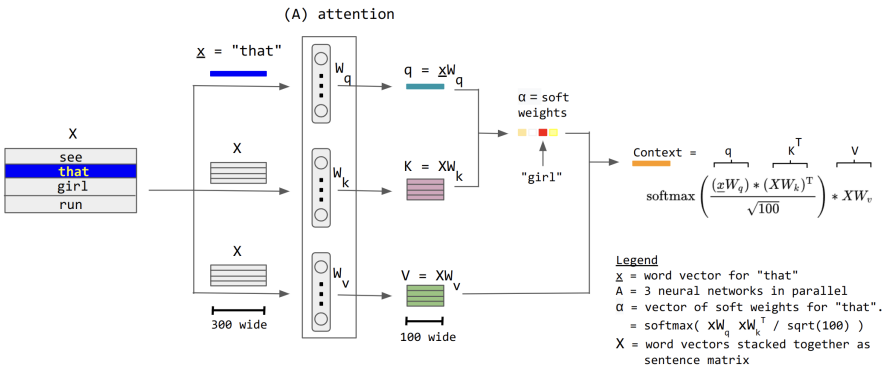



Figure 97: Self-attention: each position computes attention over all positions, enabling global context. Every position can directly attend to every other position in a single layer.

**What each projection learns:**

1. $W_Q$: How to represent "what I'm looking for"—transforms input into query space
2. $W_K$: How to represent "what I contain"—transforms input into key space for matching
3. $W_V$: How to represent "what I provide when attended to"—transforms input into value space

**Connection to kernel regression:** Self-attention can be viewed as a highly flexible kernel regression where:

- The kernel function is learned ($QK^\top$ with learned projections)
- The input representation is learned (via $W_Q$, $W_K$)
- The output representation is learned (via $W_V$)

This is extremely over-parameterised—we're simultaneously learning the similarity measure, the representations being compared, and the representation of outputs. This flexibility is both a strength (expressiveness) and a weakness (requires lots of data and regularisation).

**Why Self-Attention?**

- **Global context**: Every position can directly attend to every other position in a single layer (unlike CNNs that need many layers for global receptive fields)
- **Parallelisable**: No sequential dependencies—all positions can be computed simultaneously (unlike RNNs)
- **Flexible relationships**: Learns which relationships matter, rather than assuming fixed patterns (like convolutions' local structure)
- **Interpretable**: Attention weights show which positions influence each output

**NB!**

Self-attention has $O(n^2)$ complexity in sequence length $n$—every position attends to every other position. This becomes prohibitive for very long sequences (e.g., documents with thousands of tokens). Various "efficient attention" mechanisms exist to address this (sparse attention, linear attention, etc.), but the quadratic scaling remains a fundamental limitation.

## 122 Summary

**Key Concepts from Week 11**

1. **Gradient problems**: Vanishing (use ReLU, batch norm, residual connections); Exploding (use gradient clipping, careful initialisation)
2. **Batch normalisation**: Stabilises activations, enables higher learning rates, acts as regularisation
3. **Regularisation**: Weight decay constrains magnitudes; dropout encourages redundancy and prevents co-adaptation
4. **CNNs**: Exploit local structure via convolutions; weight sharing provides translation equivariance; pooling provides translation invariance; hierarchical feature learning builds from edges to objects
5. **Convolution**: Sparse matrix multiplication with shared, local weights—massive parameter reduction with useful inductive bias
6. **Padding/Strides**: Control output dimensions and computational cost; same padding preserves dimensions, strides reduce them
7. **RNNs**: Maintain hidden state for sequential data; suffer from gradient and capacity limitations; theoretically powerful but practically challenging
8. **Attention**: Learned weighted averaging; enables flexible, global context; soft dictionary lookup
9. **Self-attention**: Each position attends to all positions; foundation of Transformers; $O(n^2)$ complexity

**Architecture Selection Guide**

- **Spatial/local structure (images)**: CNNs
- **Sequential with short-range dependencies**: 1D CNNs or RNNs
- **Sequential with long-range dependencies**: Transformers (attention-based)
- **Variable-length sequences**: RNNs or Transformers (not vanilla MLPs)
- **Need interpretable relationships**: Attention mechanisms

# 123 Overview

**Unsupervised Learning: Core Concepts**

Unsupervised learning discovers structure in unlabelled data:

1. **Principal Component Analysis (PCA)**: Linear dimensionality reduction via variance maximisation
2. **Embedding methods**: t-SNE and UMAP for visualisation of high-dimensional data
3. **Autoencoders**: Neural network-based representation learning

The unifying theme is **compression**: finding low-dimensional representations that preserve meaningful structure.

The fundamental challenge in unsupervised learning is that we have no labels to guide us—no "correct answers" against which to measure performance. Instead, we seek to discover inherent structure in the data itself. This week focuses on **dimensionality reduction**: given high-dimensional data $X \in \mathbb{R}^{n \times D}$, find a lower-dimensional representation $Z \in \mathbb{R}^{n \times L}$ (where $L \ll D$) that preserves the essential information.
Why reduce dimensionality? Several reasons:

- **Visualisation**: Human perception is limited to 2–3 dimensions; projecting data to 2D enables visual exploration
- **Computational efficiency**: Many algorithms scale poorly with dimension; reducing $D$ accelerates training
- **Noise reduction**: High-dimensional data often contains redundant or noisy features; compression can filter these out
- **Feature learning**: The reduced representation may capture semantically meaningful factors of variation
- **Curse of dimensionality**: In high dimensions, distances become less meaningful and data becomes sparse

# 124 Principal Component Analysis

## 124.1 Dimensionality Reduction as Optimisation

PCA frames dimensionality reduction as an optimisation problem. A key insight is that although unsupervised (no labels), the structure closely resembles supervised learning—some call it "supervised learning in a trenchcoat!"

**PCA as Self-Supervised Learning**

**Central objective:**

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

This familiar form from supervised learning becomes, for dimension reduction:

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \|x_i - \text{decode}(\text{encode}(x_i, \theta), \theta)\|^2$$

**Breaking down each component:**

- **Point estimate** $\hat{\theta}$: The parameters we seek—in PCA, this is a matrix (the principal components)
- **Loss function** $\mathcal{L}(\theta)$: Measures reconstruction error—how much information is lost when compressing and decompressing
- **Optimisation** $\arg\min_\theta$: Find parameters minimising this reconstruction error
- $\text{encode}(x_i, \theta)$: Maps $x_i \in \mathbb{R}^D$ to latent representation $z_i \in \mathbb{R}^L$ with $L < D$
- $\text{decode}(z_i, \theta)$: Reconstructs the original space from the latent representation

This is effectively **auto-supervised learning**—the input serves as its own target. We predict the input from itself, but through an information bottleneck that forces compression.

The key difference from supervised learning: instead of predicting labels $y_i$ from features $x_i$, we predict $x_i$ from $x_i$ itself, via a compressed intermediate representation. The bottleneck forces the model to identify the most important factors of variation.

## 124.2 Low-Dimensional Representation

### PCA Decomposition

**Goal:** Represent each vector $x_i \in \mathbb{R}^D$ in a lower-dimensional space.
**Approximation:**

$$x_i \approx \sum_{l=1}^{L} z_{il} w_l = W z_i$$

where:

- $x_i \in \mathbb{R}^D$ is the **original high-dimensional vector**
- $z_i \in \mathbb{R}^L$ is the **latent representation**—the coordinates of $x_i$ in the reduced $L$-dimensional space
- $w_l \in \mathbb{R}^D$ are the **principal components**—basis vectors (directions) capturing maximum variance
- $W \in \mathbb{R}^{D \times L}$ is the matrix whose columns are the principal components

**Loss function (reconstruction error):**

$$\begin{aligned}\mathcal{L}(W, Z) &= \frac{1}{n} \left\| X - W Z^\top \right\|_F^2 \\ &= \frac{1}{n} \sum_{i=1}^{n} \|x_i - W z_i\|^2\end{aligned}$$

where $X$ is the $n \times D$ data matrix and $Z$ is the $n \times L$ matrix of latent representations. The Frobenius norm $\| \cdot \|_F$ measures the total squared error across all entries.

**Intuition:** PCA finds orthogonal directions (principal components) along which data varies most. The first principal component captures the direction of maximum variance; the second captures maximum variance orthogonal to the first; and so on. Projecting onto the first $L$ components provides the best $L$-dimensional linear approximation in the least-squares sense.
Think of it geometrically: if your data lies roughly along an ellipsoid in high-dimensional space, PCA finds the axes of that ellipsoid, ordered by length. The longest axis (most variance) becomes PC1, the second-longest becomes PC2, etc.

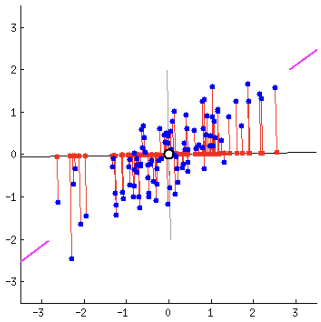



Figure 98: PCA projection: data points (blue) in a 2D space projected onto the first principal component (red line). The principal component direction captures maximum variance; projections minimise the perpendicular (orthogonal) distance to this line. The residuals (dashed lines) represent the reconstruction error.

## 124.3 Mathematical Derivation of PCA

**PCA via Eigendecomposition**

Given centred data $X$ (zero mean), PCA can be derived as follows:
**Step 1: Compute the covariance matrix**

$$\Sigma = \frac{1}{n} X^\top X \in \mathbb{R}^{D \times D}$$

**Step 2: Eigendecomposition**
Find eigenvalues $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_D$ and corresponding eigenvectors $v_1, v_2, \ldots, v_D$:

$$\Sigma v_l = \lambda_l v_l$$

**Step 3: Select top $L$ components**
The principal components are the eigenvectors corresponding to the $L$ largest eigenvalues:

$$W = [v_1 \mid v_2 \mid \ldots \mid v_L] \in \mathbb{R}^{D \times L}$$

**Step 4: Project data**
The low-dimensional representation is:

$$Z = XW \in \mathbb{R}^{n \times L}$$

**Variance explained:**
The proportion of variance explained by the first $L$ components is:

$$\frac{\sum_{l=1}^{L} \lambda_l}{\sum_{l=1}^{D} \lambda_l}$$

**NB!**

**Centering is essential!** PCA assumes zero-mean data. If you don't centre your data (subtract the mean), the first principal component will point toward the mean rather than capturing variance. Always preprocess: $\tilde{X} = X - \bar{X}$.
Additionally, if features have very different scales (e.g., one ranges 0–1, another 0–10000), PCA will be dominated by the high-variance feature. Consider standardising (dividing by standard deviation) when scales differ substantially.

### 124.4 Terminology: Embeddings

**Embeddings**

An **embedding** is a mapping from high-dimensional data to a lower-dimensional space that preserves relevant structure.
Embeddings appear throughout machine learning:

- **Word embeddings** (Word2Vec, GloVe): words → dense vectors capturing semantic relationships ("king" - "man" + "woman" ≈ "queen")
- **Graph embeddings**: nodes → vectors preserving graph structure (connected nodes map to nearby vectors)
- **Image embeddings**: images → feature vectors from CNN intermediate layers
- **Sentence embeddings**: sentences → vectors capturing meaning (BERT, sentence transformers)

PCA produces **linear** embeddings; neural methods (autoencoders, t-SNE) learn **non-linear** embeddings that can capture more complex manifold structure.

The term "embedding" emphasises that we're not just reducing dimensions but *embedding* data into a space where geometric relationships (distances, angles) carry semantic meaning.

## 125 Neighbourhood Embeddings

PCA preserves **global** structure via linear projections—it maintains large-scale geometric relationships. However, for visualisation, we often want methods that preserve **local** structure—keeping nearby points together even if global distances are distorted. This is particularly useful for discovering clusters.

## 125.1 Stochastic Neighbour Embedding (SNE)

### SNE Probability Model

**Core idea:** Convert pairwise distances into probabilities representing "neighbour" relationships, then find a low-dimensional embedding where these neighbourhood probabilities are preserved.

**Step 1: High-dimensional similarities**

For each pair of points, compute the probability that $x_i$ would "pick" $x_j$ as a neighbour:

$$p_{j|i} = \frac{\exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma_i^2}\right)}{\sum_{k \neq i} \exp\left(-\frac{\|x_i - x_k\|^2}{2\sigma_i^2}\right)}$$

This is a softmax over squared distances: points close to $x_i$ get high probability; distant points get low probability. The bandwidth $\sigma_i$ controls how "local" the neighbourhood is (set adaptively per point).

**Step 2: Low-dimensional similarities**

Define analogous probabilities in the embedding space:

$$q_{j|i} = \frac{\exp\left(-\|z_i - z_j\|^2\right)}{\sum_{k \neq i} \exp\left(-\|z_i - z_k\|^2\right)}$$

where $z_i, z_j \in \mathbb{R}^2$ (or $\mathbb{R}^3$) are the low-dimensional representations.

**Step 3: Minimise divergence**

Find embeddings $Z$ that make $Q$ as similar to $P$ as possible:

$$\mathcal{L} = \sum_i D_{\mathrm{KL}}(P_i \| Q_i) = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}$$

The KL divergence penalises cases where $p_{j|i}$ is high but $q_{j|i}$ is low (nearby points in high dimensions that are far apart in the embedding).

**Interpretation:** We're asking: "If I randomly selected neighbours for each point according to a Gaussian kernel, would the neighbourhood structure look the same in high and low dimensions?" SNE optimises the embedding to make this true.

## 125.2 t-Distributed SNE (t-SNE)

**t-SNE: Addressing SNE's Limitations**

**Problem with SNE:** The Gaussian distribution has light tails. This creates an asymmetric cost structure:

- **Cheap:** Placing distant high-dimensional points close together in the embedding (low $p_{j|i}$ means small penalty even if $q_{j|i}$ is large)
- **Expensive:** Separating nearby high-dimensional points in the embedding (high $p_{j|i}$ creates large penalty if $q_{j|i}$ is small)

This causes the "crowding problem": SNE tends to crush distant clusters together rather than spreading them out.
**Solution:** Use the Student's t-distribution (with 1 degree of freedom, i.e., Cauchy distribution) for low-dimensional similarities:

$$q_{j|i} = \frac{\left(1 + \|z_i - z_j\|^2\right)^{-1}}{\sum_{k \neq i} \left(1 + \|z_i - z_k\|^2\right)^{-1}}$$

The t-distribution has **heavier tails** than the Gaussian. This means:

- Distant points in high dimensions can be modelled as genuinely far apart in the embedding
- The heavy tails allow more "space" for different clusters to spread out
- Better cluster separation in visualisations

**Same objective function:**

$$\mathcal{L} = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}$$

Why specifically 1 degree of freedom? The t-distribution with 1 d.f. (Cauchy) has tails that decay as $1/r^2$ in 2D, which matches the rate at which "available area" grows with distance. This creates a natural balance for 2D embeddings.

**NB!**

**t-SNE Limitations:**

- **Computational cost:** $O(N^2)$ complexity—every point interacts with every other point (an "$N$-body problem"). Limits scalability to datasets with tens of thousands of points. (Barnes-Hut approximations can reduce this to $O(N \log N)$.)
- **Non-parametric:** No learned function to embed new points; must re-run on entire dataset
- **Hyperparameter sensitivity:** Results depend heavily on **perplexity** (effective number of neighbours), which must be tuned
- **Primarily for visualisation:** The embedding doesn't preserve meaningful distances; only local neighbourhood structure is reliable
- **Stochasticity:** Different runs give different results; always run multiple times

## 125.3 UMAP: Uniform Manifold Approximation and Projection

**UMAP: Efficient Manifold Learning**

**Uniform Manifold Approximation and Projection** addresses t-SNE's limitations while achieving similar (often better) visualisation quality.
**Key innovations:**

1. $k$**-nearest neighbour graph**: Instead of computing all $O(N^2)$ pairwise distances, construct a sparse graph connecting each point to its $k$ nearest neighbours. This reduces complexity dramatically.

2. **Local metric adaptation**: Each point has its own distance scale, automatically adapting to varying local densities. Dense regions use small scales; sparse regions use large scales.

3. **Cross-entropy objective**: Optimises binary cross-entropy on edge existence ("is there an edge between $i$ and $j$?") rather than KL divergence. This is more symmetric and computationally efficient.

4. **Theoretical foundation**: UMAP is grounded in topological data analysis and Riemannian geometry (though the practical algorithm doesn't require understanding these).

**Result:** Near-linear scaling $O(N \log N)$ with dataset size while preserving both local and (to some extent) global structure. UMAP also produces a **parametric** model that can embed new points.

**UMAP's approach:** Rather than thinking about probability distributions over neighbours, UMAP thinks about **graph structures**. It builds a weighted graph in high dimensions (where edge weights reflect similarity) and then optimises a low-dimensional layout that preserves this graph structure.

### 125.3.1 Visual Intuition for UMAP

UMAP starts with a k-nearest neighbour graph to capture local relationships and then seeks to maintain these relationships in the low-dimensional space, ensuring that the manifold's geometry is preserved.

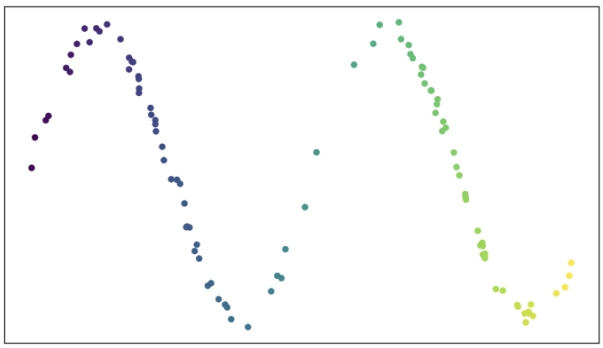


Figure 99: Original high-dimensional data projected to 2D, with colours indicating cluster membership. The goal is to find a 2D embedding that reveals this cluster structure.

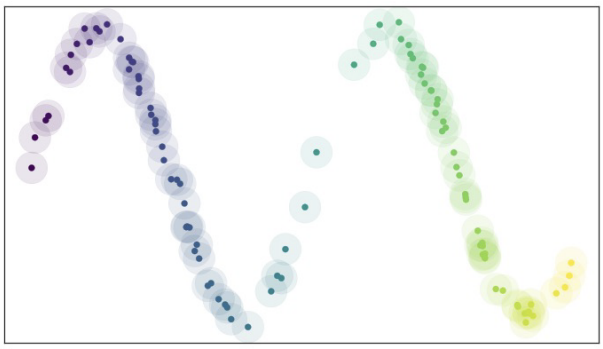


Figure 100: UMAP constructs a $k$-nearest neighbour graph; shaded circles represent local neighbourhoods. Each point is connected to its closest neighbours, capturing local structure without computing all pairwise distances.

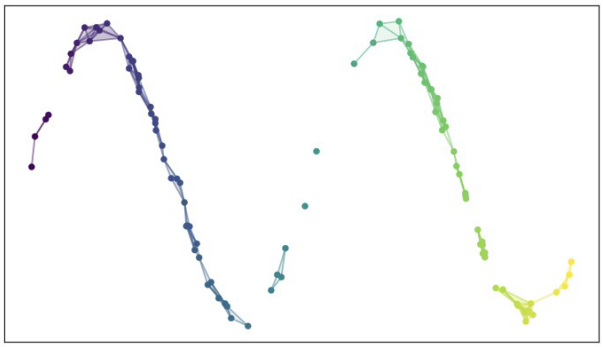


Figure 101: The embedding preserves neighbourhood structure: connected points in high dimensions remain close in the projection. The lines show how local connectivity is maintained.

### 125.3.2 Handling Varying Local Density

Real data often exhibits different local structures—some regions are dense, others sparse. A global distance threshold would either miss structure in dense regions or create spurious connections in sparse regions.

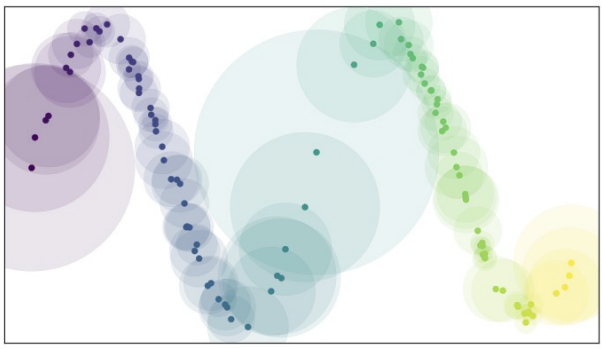


Figure 102: UMAP adapts to varying local densities, using different distance scales in dense versus sparse regions. The varying shaded regions show how neighbourhood size adapts to local density, preserving structure in both dense and sparse areas.

UMAP handles this by giving each point its own distance scale, automatically determined by the distance to its $k$-th nearest neighbour. This local adaptation is crucial for real-world data with heterogeneous density.

**t-SNE/UMAP Interactive Demo**

For building intuition about how t-SNE and UMAP behave with different hyperparameters and data structures, see:
`https://pair-code.github.io/understanding-umap/`
Experimenting with the perplexity (t-SNE) and n_neighbours (UMAP) parameters on different datasets is highly recommended.

### 125.4 Comparing Dimensionality Reduction Methods

**PCA vs t-SNE vs UMAP**

| Method | Structure Preserved | Complexity | Use Case |
|---|---|---|---|
| PCA | Global, linear | $O(D^2 n)$ | Pre-processing, interpretable axes, when linear structure suffices |
| t-SNE | Local, non-linear | $O(n^2)$ | Visualisation (small–medium data), cluster discovery |
| UMAP | Local + some global | $O(n \log n)$ | Visualisation (large data), preserves more global structure |

**When to use which:**

- **PCA**: When you need interpretable components, when data has linear structure, as a preprocessing step before other algorithms, when you need to embed new points easily
- **t-SNE**: When you want beautiful cluster visualisations, when local structure matters more than global, for exploratory data analysis on moderately-sized datasets
- **UMAP**: When t-SNE is too slow, when you want to preserve more global structure, when you need to embed new points, as a general-purpose alternative to t-SNE

**NB!**

**Interpreting t-SNE/UMAP plots:**

- **Cluster sizes are meaningless**: A visually larger cluster doesn't mean more data points
- **Distances between clusters are unreliable**: Two clusters appearing far apart may not be far in the original space
- **Only local neighbourhoods are trustworthy**: Points that are close together in the embedding were likely close in high dimensions
- **Run multiple times**: Results are stochastic; consistent patterns across runs are more reliable

## 126 Autoencoders

Autoencoders learn non-linear dimensionality reduction using neural networks. The key idea is **self-supervision**: predict the input from itself via a compressed intermediate representation.

This extends PCA's linear compression to arbitrary non-linear transformations.

## 126.1 Architecture

**Autoencoder Architecture**

An autoencoder consists of three parts:

1. **Encoder** $f_\phi : \mathbb{R}^D \to \mathbb{R}^L$ compresses input to latent space
2. **Bottleneck** layer with $L < D$ dimensions—the learned representation
3. **Decoder** $g_\psi : \mathbb{R}^L \to \mathbb{R}^D$ reconstructs input from latent code

**Objective:**

$$\hat{\phi}, \hat{\psi} = \arg\min_{\phi,\psi} \frac{1}{n} \sum_{i=1}^{n} \|x_i - g_\psi(f_\phi(x_i))\|^2$$

where $\phi$ and $\psi$ are the parameters (weights) of the encoder and decoder networks respectively.

**Training via backpropagation:**

1. Forward pass: $x_i \to z_i = f_\phi(x_i) \to \hat{x}_i = g_\psi(z_i)$
2. Compute loss: $\mathcal{L}_i = \|x_i - \hat{x}_i\|^2$
3. Backward pass: Compute gradients $\nabla_\phi \mathcal{L}$ and $\nabla_\psi \mathcal{L}$
4. Update: $\phi \leftarrow \phi - \eta \nabla_\phi \mathcal{L}$, $\psi \leftarrow \psi - \eta \nabla_\psi \mathcal{L}$

The bottleneck forces the network to learn a compressed representation capturing the most important features for reconstruction.

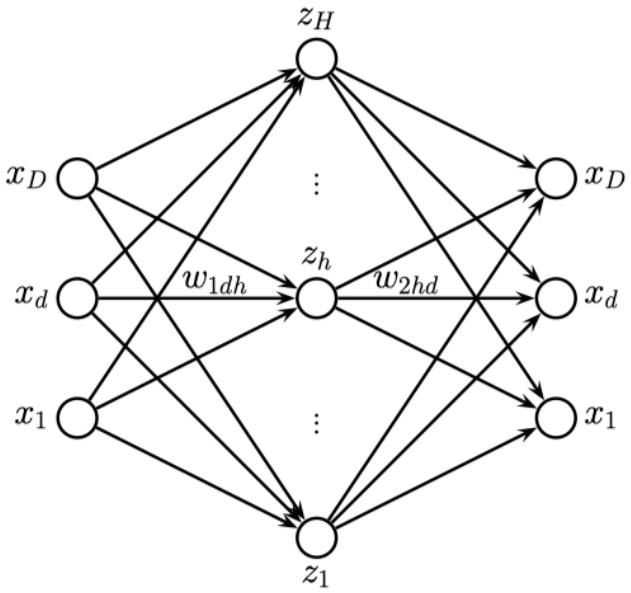



Figure 103: Autoencoder architecture: input is compressed through the encoder to a bottleneck layer (latent representation $z$), then reconstructed by the decoder. The bottleneck dimension $L$ is smaller than the input dimension $D$, forcing compression.

**Intuition:** The encoder must learn to "distil" the input into a compressed code that retains enough information for the decoder to reconstruct. If $L$ is small, only the most important features can be preserved. The network learns which features matter most for reconstruction.

## 126.2 Relationship to PCA

**Linear Autoencoders Recover PCA**

Under specific conditions, autoencoders are exactly equivalent to PCA:

- Single hidden layer (one encoder layer, one decoder layer)
- **Linear activations** (no non-linearities: no ReLU, sigmoid, etc.)
- Mean squared error loss
- Encoder and decoder share (transposed) weights

In this case, the encoder learns to project onto the principal component subspace, and the decoder learns to reconstruct from this projection. The learned weights span the same subspace as the top $L$ principal components.
**With non-linearities**, autoencoders can learn **non-linear manifolds** that PCA cannot capture. A deep autoencoder with multiple hidden layers can learn hierarchical features, with each layer capturing increasingly abstract representations.

This equivalence is profound: it shows that PCA is a special case of autoencoders when we restrict to linear transformations. Adding non-linearities strictly increases expressiveness.

### 126.3 Bottleneck Dimension

**NB!**

**The bottleneck is crucial:**
**If $L \geq D$:** The autoencoder can learn the identity function—simply copying inputs without learning meaningful structure. There's no compression, so no useful representation is learned.
**If $L$ is too small:** The bottleneck may not have enough capacity to represent the data, leading to poor reconstructions and loss of important information.
**Even with $L < D$:** An overly expressive network (too many layers, too many parameters) might learn trivial solutions or overfit to noise. Regularisation helps ensure meaningful representations:

- Weight decay (L2 regularisation on weights)
- Dropout in encoder/decoder
- Early stopping based on validation reconstruction error

Choosing the right bottleneck dimension $L$ is a hyperparameter. Strategies include:

- Start with PCA variance analysis: if 95% of variance is in 10 components, try $L \approx 10$
- Monitor reconstruction quality on held-out data
- For visualisation, $L = 2$ or $L = 3$

### 126.4 Autoencoder Variants

Different regularisation strategies lead to different autoencoder variants, each suited to different tasks:

**Denoising Autoencoders**

**Idea:** Train the network to be robust to noise by corrupting inputs and asking it to reconstruct the *clean* version.
**Training procedure:**

1. Add noise to input: $\tilde{x}_i = x_i + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$ (or use dropout noise, masking random features)
2. Train to reconstruct original: $\min \|x_i - g_\psi(f_\phi(\tilde{x}_i))\|^2$

**Why it works:**

- Forces representations to be invariant to small perturbations
- Cannot simply memorise inputs (the noise varies each time)
- Learns robust features that capture true structure, not noise
- Particularly useful when data is noisy

## Sparse Autoencoders

**Idea:** Encourage the latent representation to be **sparse**—most entries should be zero (or near-zero) for any given input.
**Objective:**

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} \|x_i - g_\psi(f_\phi(x_i))\|^2 + \lambda\|z_i\|_1$$

The L1 penalty $\|z_i\|_1 = \sum_l |z_{il}|$ encourages sparsity.
**Why it's useful:**

- Each latent dimension tends to represent a distinct "feature" of the data
- Sparse representations are often more interpretable
- Analogous to L1 regularisation in linear models (Lasso)
- Can use $L > D$ if sparsity is enforced (overcomplete representations)

## Variational Autoencoders (VAEs)

**Idea:** Make the autoencoder probabilistic by having the encoder output *distribution parameters* rather than point estimates.
**Architecture:**

1. Encoder outputs parameters: $f_\phi(x_i) = (\mu_i, \sigma_i^2)$
2. Sample latent code: $z_i \sim \mathcal{N}(\mu_i, \text{diag}(\sigma_i^2))$
3. Decoder reconstructs: $\hat{x}_i = g_\psi(z_i)$

**Objective (Evidence Lower Bound / ELBO):**

$$\mathcal{L} = \underbrace{\mathbb{E}_{z\sim q_\phi(z|x)}[\log p_\psi(x|z)]}_{\text{reconstruction}} - \underbrace{D_{\text{KL}}(q_\phi(z|x)\|p(z))}_{\text{regularisation}}$$

The KL term encourages the learned latent distribution $q_\phi(z|x)$ to be close to a standard normal prior $p(z) = \mathcal{N}(0, I)$.
**Key property: Generation**
Because the latent space is regularised to be Gaussian, we can **generate new data**:

1. Sample $z \sim \mathcal{N}(0, I)$
2. Decode: $x_{\text{new}} = g_\psi(z)$

This produces novel samples that look like the training data.

**When to Use Each Variant**

- **Standard autoencoder**: Feature learning, pre-training, anomaly detection (high reconstruction error $\Rightarrow$ anomaly)
- **Denoising**: When data is noisy, when you want robust features, as a regularisation technique
- **Sparse**: When you want interpretable, disentangled representations; feature extraction
- **Variational (VAE)**: Generative modelling, sampling new data points, learning smooth latent spaces, semi-supervised learning

## 126.5 Applications of Autoencoders

**Anomaly detection:** Train an autoencoder on "normal" data. At test time, compute reconstruction error for new samples. High reconstruction error indicates the sample is unlike the training data—potentially anomalous.
**Pretraining:** Use the encoder from a trained autoencoder as initialisation for a supervised task. The encoder has learned useful features from unlabelled data.
**Denoising:** Train a denoising autoencoder, then use it to clean noisy inputs (e.g., removing noise from images).
**Compression:** The latent code $z$ is a compressed representation of $x$. This can be used for data compression (though specialised codecs typically outperform).

# 127 Summary

**Week 12: Key Concepts**

**Dimensionality Reduction:**

- **PCA**: Linear projection maximising variance; optimal for Gaussian data; interpretable axes; equivalent to linear autoencoder
- **Autoencoders**: Non-linear compression via neural networks; can learn complex manifold structure

**Visualisation Methods:**

- **t-SNE**: Preserves local neighbourhood structure using heavy-tailed t-distribution; excellent for cluster discovery; $O(N^2)$
- **UMAP**: Scalable alternative using $k$-NN graphs; preserves more global structure; $O(N \log N)$

**Core Principles:**

- Unsupervised learning seeks structure without labels—the data is its own teacher
- Compression (bottlenecks) forces models to identify and capture essential features
- Linear methods (PCA) are interpretable but limited; non-linear methods capture complex structure
- Choice of method depends on goal: interpretability vs flexibility, global vs local structure, visualisation vs downstream learning

**Connections:**

- Linear autoencoder $\equiv$ PCA (under specific conditions)
- VAE decoder enables generative modelling (connects to probabilistic graphical models)
- Embeddings from unsupervised pre-training improve downstream supervised tasks (transfer learning)
- t-SNE/UMAP reveal cluster structure that can guide further analysis or labelling

**The big picture:** Unsupervised learning is about finding structure. PCA finds linear structure (directions of variance). t-SNE/UMAP find neighbourhood structure (clusters). Autoencoders find reconstruction-relevant structure (features that matter for predicting the input). Each perspective reveals different aspects of the data, and combining them often provides the most insight.