Foundations of Machine Learning

Lecture Notes by Dat Quoc Truong

Chapters 1 – 10: From Linear Regression to Reinforcement Learning

A Comprehensive Beginner's Textbook

Contents

Chapter 1

Introduction to Machine Learning

This chapter introduces the foundations of machine learning. We begin with a high-level overview that motivates why machine learning matters and where it appears in modern technology. We then distinguish supervised from unsupervised learning, the two largest families of learning algorithms. After that, we focus on the simplest yet most foundational supervised learning algorithm — *linear regression with one variable* — and study its model, its cost function, and its visualisation. Finally, we examine *gradient descent*, the optimisation algorithm used to train the model from data.

1.1 Overview of Machine Learning

1.1.1 Why Machine Learning Matters

Machine learning (ML) is a foundational technology behind countless applications that shape modern life: web search engines, email spam filtering, speech recognition, medical diagnosis, recommendation systems, autonomous driving, and many more. Its strength lies in solving problems that are difficult or impossible to express with explicit rules, by instead *learning patterns directly from data*.

Machine learning is widely considered a sub-field of artificial intelligence (AI). It is sometimes confused with the more ambitious goal of *Artificial General Intelligence* (AGI), which refers to systems with broad, human-level cognitive abilities. AGI remains largely hypothetical and is often overhyped in popular media. Practical ML, in contrast, is a mature engineering discipline with concrete tools and well-understood algorithms.

1.1.2 What Is Machine Learning?

The basic idea of machine learning is captured in a simple sentence: *learn to perform a task from data*. A more technical phrasing is that an ML system *recognises and extracts patterns* from data, and then uses those patterns to make predictions, decisions, or descriptions about new, unseen inputs.

A classical definition was given by Tom Mitchell (1997):

> *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance on tasks in T, as measured by P, improves with experience E.*

This definition highlights three key ingredients: **experience** (data the system learns from), **tasks** (what the system must do), and **performance measure** (how we evaluate success).

1.1.3 Major Families of ML Algorithms

Machine learning algorithms are commonly grouped into the following families:

- **Supervised learning.** The most widely used family in real-world applications. Each training example is a pair: an input x and a corresponding correct output y. The algorithm learns a mapping $f : x \to y$.
- **Unsupervised learning.** The data consists only of inputs x, with no labels. The goal is to discover hidden structure — for example, clusters, low-dimensional patterns, or anomalies.
- **Recommender systems.** A specialised family that predicts a user's preferences for items, used by streaming services, e-commerce platforms, and social media.
- **Reinforcement learning.** An agent learns by interacting with an environment, receiving *rewards* or *penalties*, and gradually improving its behaviour to maximise cumulative reward.

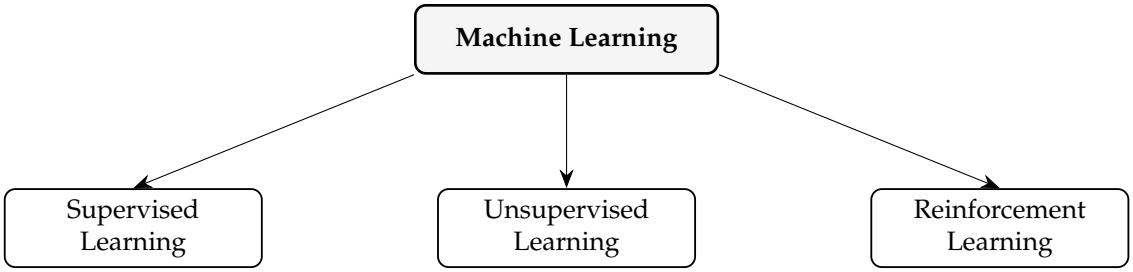


Figure 1.1: Three major families of machine learning algorithms. Recommender systems are a specialised sub-area of supervised and unsupervised learning.

1.2 Supervised vs. Unsupervised Machine Learning

1.2.1 Supervised Learning

In supervised learning, every training example has the form (x, y), where x is the *input* (also called a *feature* or *feature vector*) and y is the correct *output* (also called the *label* or *target*). The objective is to learn a function f such that $f(x) \approx y$ on new, unseen inputs. The accuracy of the resulting model depends on the difficulty of the task, the amount and quality of the dataset, and the choice of algorithm.

Supervised learning problems split into two broad categories depending on the type of the output variable.

Classification problems. The output y is *discrete*, i.e. a category label. Examples include:

- **Binary** (2 classes): yes/no, 0/1, true/false. *Example: Is this email spam?*
- **Multiclass** (≥ 3 classes): classifying a fruit as *apple*, *mandarin*, or *lemon* based on width and height.
- **Multilabel**: an example may belong to several classes at once. *Example: tagging a photo with both "beach" and "sunset."*

Common practical classification tasks include:

1. Detecting whether a new email is spam from its text content.
2. Predicting whether a user will buy a product, given features such as age, gender, and browsing history.
3. Forecasting whether tomorrow will be rainy, using humidity, wind speed, and cloudiness.
4. Detecting whether a credit-card transaction is fraudulent.

Regression problems. The output y is *continuous* (a real number). The model predicts a real value $\hat{y} = f(x)$, attempting to be as close to the true y as possible. Examples include predicting house price from size, temperature from time of day, or a patient's blood pressure from lifestyle factors.

Figure 1.2 contrasts the two types visually.

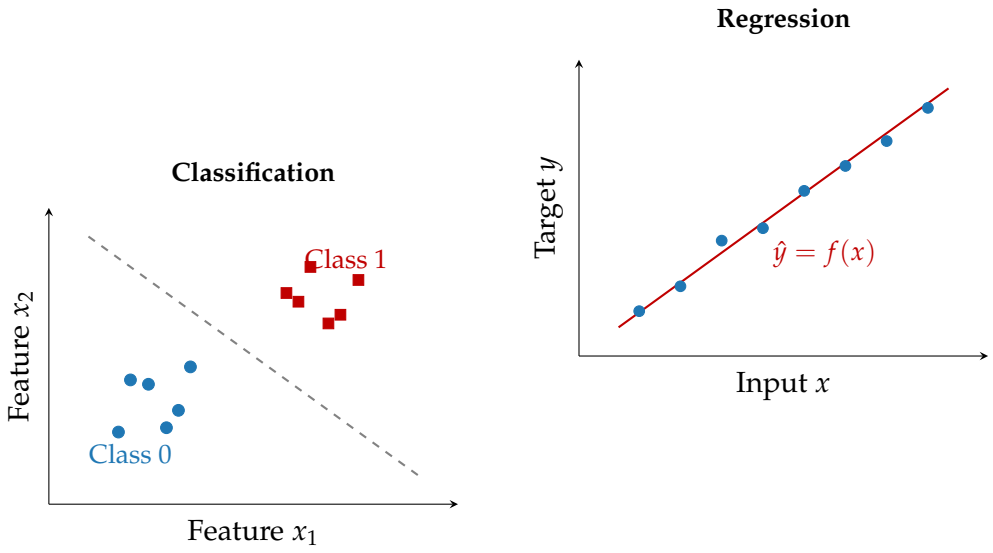


Figure 1.2: Left: classification separates discrete categories with a decision boundary. Right: regression fits a continuous function to the data.

1.2.2 Unsupervised Learning

In unsupervised learning, the dataset consists only of inputs $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$, with *no* corresponding labels y. The algorithm must discover useful structure entirely on its own, without any guidance from human-provided answers.

Common types of unsupervised learning include:

- **Clustering.** Group similar data points together. *Examples:* grouping news articles by topic; grouping customers by purchasing behaviour; analysing DNA microarrays to find gene patterns.
- **Anomaly detection.** Find unusual data points that deviate from normal behaviour, e.g. unusual credit-card transactions for fraud monitoring.

- **Dimensionality reduction.** Compress the data by representing it with fewer numbers while preserving important structure. Useful for visualisation and pre-processing.

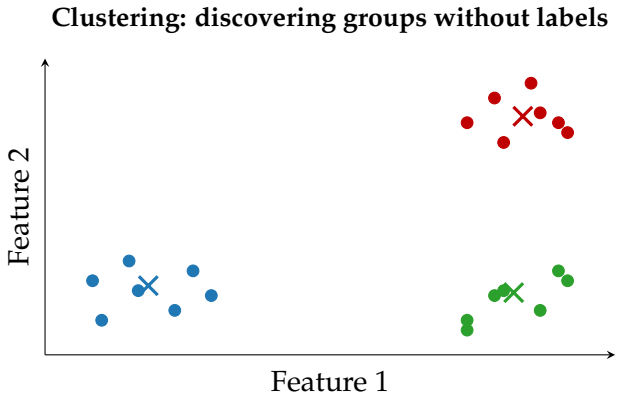


Figure 1.3: A clustering algorithm assigns points to three groups (shown by colour) without ever being told the correct group. Crosses mark the centroids.

1.2.3 Side-by-Side Comparison

Aspect	Supervised learning	Unsupervised learning
Training data	Pairs (x, y) with labels	Inputs x only, no labels
Goal	Learn f s.t. $f(x) \approx y$	Discover hidden structure
Typical tasks	Classification, regression	Clustering, anomaly detection, dimensionality reduction
Classic example	Predicting house price from size	Grouping customers by behaviour

1.3 Linear Regression Model

We now zoom in on the simplest and most important supervised learning model: **linear regression with one variable**. Linear regression fits a *straight line* to a dataset and uses that line to predict numerical outputs from a single numerical input.

1.3.1 Setup and Notation

Suppose we wish to predict house prices from house sizes. A typical training set might look like:

Size in ft^2 (x, feature)	Price in \$1000s (y, target)
2104	460
1416	232
1534	315
852	178
⋮	⋮

We adopt the following standard notation:

Symbol	Name	Meaning
m	Number of examples	Total rows in the dataset
x	Input / feature	E.g. house size in ft^2
y	Output / target	E.g. house price in \$1000s
(x, y)	Training example	A single (input, output) pair
$(x^{(i)}, y^{(i)})$	i-th example	Index i is a row index, *not* an exponent
$\hat{y}$	Prediction	Model output for a given x

1.3.2 The Linear Model

Linear regression assumes the relationship between x and y is approximately a straight line:

$$\hat{y} \;=\; f_{w,b}(x) \;=\; w\,x + b, \tag{1.1}$$

where w is the *slope* (or *weight*) and b is the *intercept* (or *bias*). Together, w and b are the **parameters** of the model — the variables we adjust during training. Because y is a real number and the model's output is also a real number, this is a *regression* model (not classification).

1.3.3 The Cost Function

To choose good parameters w and b we need to measure how well the line fits the data. This measure is called the **cost function** or *loss function*. For linear regression the standard choice is the **mean squared error (MSE)** cost:

$$J(w,\, b) \;=\; \frac{1}{2m}\sum_{i=1}^{m}\big(f_{w,b}(x^{(i)}) - y^{(i)}\big)^2 \;=\; \frac{1}{2m}\sum_{i=1}^{m}\big(w\,x^{(i)} + b - y^{(i)}\big)^2. \tag{1.2}$$

The factor $\frac{1}{m}$ averages over all training examples, and the extra $\frac{1}{2}$ is a convenience that makes differentiation cleaner (the 2 from the power rule cancels). The **goal** is to choose w, b to *minimise* $J(w, b)$: when J is small, the model's predictions are close to the true targets.

1.3.4 Visualising the Cost Function

To build intuition, temporarily fix $b = 0$ so the model becomes $f_w(x) = wx$. As we vary w, the line rotates around the origin, and $J(w)$ traces a smooth, bowl-shaped (convex) curve.

When both w and b are free, $J(w, b)$ is a bowl in three dimensions. We can visualise it as a *contour plot*: closed curves in the (w, b)-plane where J is constant.

Worked example. Consider the tiny training set $\{(1,1), (2,2), (3,3)\}$ with model $f_w(x) = wx$ ($b = 0$).

- $w = 1$: predictions match targets exactly $\Rightarrow J(1) = 0$.
- $w = 0.5$: predictions are 0.5, 1.0, 1.5; squared errors are 0.25, 1.0, 2.25; so $J(0.5) =$

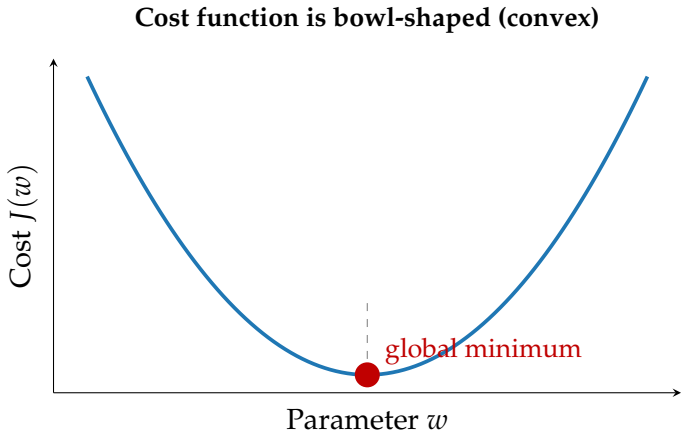


Figure 1.4: For fixed b, the cost $J(w)$ is a convex (bowl-shaped) function with a unique global minimum. This guarantees gradient descent will find the optimal w.

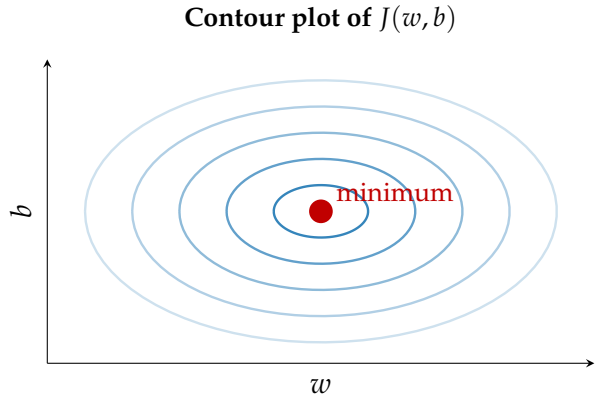


Figure 1.5: Contour plot of $J(w, b)$. Each elliptical curve is a level set $J(w, b) = c$ for some constant c. The global minimum is at the centre.

$\frac{3.5}{6} \approx 0.583$.

- $w = 0$: predictions all zero; $J(0) = \frac{14}{6} \approx 2.33$.

Among these three choices, $w = 1$ is optimal — perfectly fitting the data.

1.4 Training the Model with Gradient Descent

We now have a clear objective: find (w, b) that minimises $J(w, b)$. For linear regression with one variable, a closed-form solution exists, but for almost all other ML models we must rely on numerical optimisation. The most important such algorithm is **gradient descent**.

1.4.1 The Idea

Imagine standing at some point on a hilly landscape that represents the cost surface. Your goal is to reach the lowest valley. A sensible local strategy is: *look in every direction, find where the ground slopes most steeply downward, and take a small step that way.* Repeat until you can no longer go down. This is precisely what gradient descent does. At each iteration it evaluates the *gradient* (vector of partial derivatives) at the current point and moves a small amount in the direction *opposite* to the gradient.

Algorithm outline.

1. **Initialise** the parameters, typically $w = 0$, $b = 0$.
2. **Repeat until convergence:** update w and b in the direction that most reduces $J(w, b)$.
3. **Stop** when the parameters change very little between iterations (the algorithm has reached a local, or for convex costs, the global minimum).

1.4.2 The Update Rule

The gradient descent update rule is applied *simultaneously* to all parameters at every iteration:

$$w \leftarrow w - \alpha \frac{\partial J(w,b)}{\partial w}, \tag{1.3}$$

$$b \leftarrow b - \alpha \frac{\partial J(w,b)}{\partial b}. \tag{1.4}$$

Here $\alpha > 0$ is the **learning rate**, a small positive number that controls the size of each step. The word *simultaneous* is important: both updates must use the *old* values of w and b. The correct implementation is:

$$\begin{aligned} \text{tmp}_w &= w - \alpha \frac{\partial J}{\partial w}(w,b), \\ \text{tmp}_b &= b - \alpha \frac{\partial J}{\partial b}(w,b), \\ w &\leftarrow \text{tmp}_w, \quad b \leftarrow \text{tmp}_b. \end{aligned}$$

1.4.3 Derivative Intuition

For a simplified one-variable setting ($b = 0$), the update is $w \leftarrow w - \alpha \frac{dJ}{dw}$.

- If the slope $\frac{dJ}{dw} > 0$ (cost increases with w), the rule *decreases* w — moving toward the minimum.
- If the slope $\frac{dJ}{dw} < 0$ (cost decreases with w), the rule *increases* w — again moving toward the minimum.

1.4.4 Choosing the Learning Rate

The learning rate α has a profound effect on training behaviour.

Too small: Each step is tiny, so convergence is very slow and may require an impractical number of iterations to reach the minimum.

Too large: Steps may *overshoot* the minimum, causing the cost to oscillate or even grow — the algorithm *diverges*.

Near a minimum. At a local minimum the gradient is zero, so the update term $\alpha \cdot 0 = 0$ and parameters stop changing. Moreover, as the algorithm approaches the minimum, the gradient magnitude *decreases*, so steps automatically become smaller even with a fixed α. This means gradient descent can converge with a fixed learning rate.

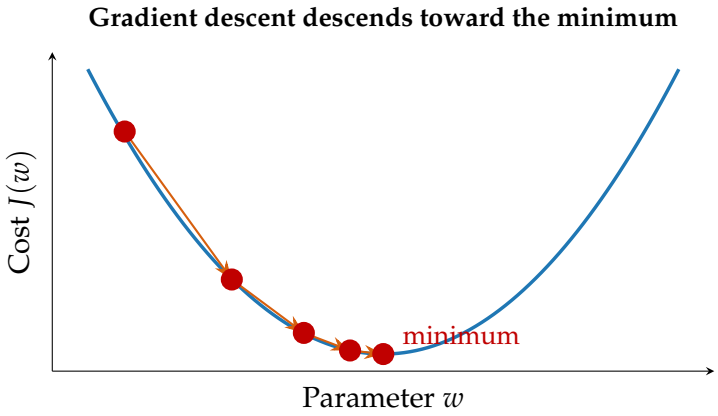


Figure 1.6: Successive gradient descent steps (orange arrows) descend toward the minimum of the cost function.

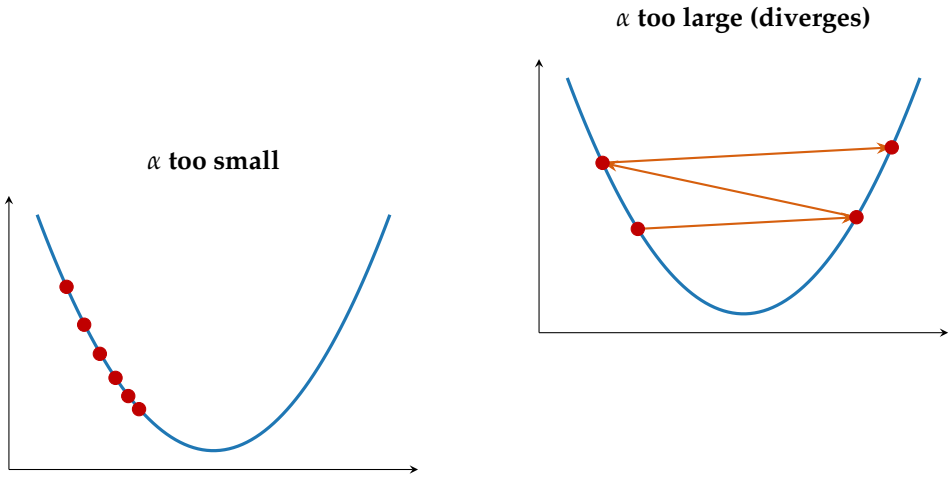


Figure 1.7: Left: a too-small α leads to painfully slow convergence. Right: a too-large α causes overshooting and divergence.

1.4.5 Gradient Descent for Linear Regression

Differentiating the MSE cost (1.2) term by term:

$$\frac{\partial J}{\partial w} = \frac{1}{m}\sum_{i=1}^{m}\left(f_{w,b}(x^{(i)}) - y^{(i)}\right)x^{(i)}, \tag{1.5}$$

$$\frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\left(f_{w,b}(x^{(i)}) - y^{(i)}\right). \tag{1.6}$$

(The factor of 2 from differentiating the square cancels with the $\frac{1}{2}$ in front of the cost.) The full **batch gradient descent for linear regression** algorithm is then:

Repeat until convergence:

$$w \leftarrow w - \frac{\alpha}{m} \sum_{i=1}^{m} \left(f_{w,b}(x^{(i)}) - y^{(i)} \right) x^{(i)},$$

$$b \leftarrow b - \frac{\alpha}{m} \sum_{i=1}^{m} \left(f_{w,b}(x^{(i)}) - y^{(i)} \right).$$

Why "batch"? This variant uses *all* m training examples to compute each gradient update (all m form a single "batch"). Other variants — stochastic gradient descent, mini-batch gradient descent — use one example or a small subset at a time and are introduced in later chapters.

Convexity guarantee. For the squared-error linear regression cost, $J(w, b)$ is a *convex* function — its 3D shape is a single bowl with no local minima other than the global minimum. Consequently, gradient descent with a sufficiently small α is *guaranteed* to converge to the global minimum regardless of where the parameters are initialised.

Chapter Summary

- Machine learning is the discipline of automatically learning patterns from data, rather than hand-coding explicit rules.
- The two largest families are **supervised learning** (labelled data, predict y from x) and **unsupervised learning** (unlabelled data, discover hidden structure).
- Supervised problems split into **classification** (discrete output) and **regression** (continuous output).
- Linear regression models the relationship as a straight line $f_{w,b}(x) = wx + b$.
- The **squared-error cost** $J(w, b)$ measures how well the line fits the data; the goal is to minimise J.
- **Gradient descent** iteratively updates w and b by moving in the direction of steepest descent. The learning rate α controls the step size.
- For squared-error linear regression, J is convex, so gradient descent always reaches the global minimum.

Exercises

1. For each of the following tasks, state whether it is *classification* or *regression*, and justify your answer.
   (a) Predicting tomorrow's maximum temperature in Celsius.
   (b) Recognising which handwritten digit (0–9) appears in an image.
   (c) Estimating the monthly rental price of an apartment.
   (d) Detecting whether a tumour is malignant or benign.

2. Given the dataset $\{(1,2),(2,4),(3,6)\}$ and the model $f_w(x) = wx$, compute $J(w)$ for $w \in \{1, 1.5, 2, 2.5\}$. Which value of w has the lowest cost? Does this make sense intuitively?

3. Suppose at iteration t the gradient $\partial J/\partial w = -3$ and $\alpha = 0.1$. By how much does w change, and in which direction?

4. Explain in your own words why using a very large learning rate (e.g. $\alpha = 100$) is problematic. What would the learning curve look like?

5. True or False: "In supervised learning, the algorithm has to figure out which categories exist by itself." Justify your answer.

6. Sketch the cost function $J(w)$ for the dataset $\{(1,3),(2,5),(3,7)\}$ and model $f_w(x) = wx$. Where is the minimum?

Chapter 2

Linear Regression with Multiple Variables

In Chapter 1 we studied simple linear regression with a *single* input feature x. Real-world prediction problems almost always involve *more than one* input variable. This chapter generalises the model so that the target y is expressed as a linear combination of n features $x_1, x_2, \ldots, x_n$. We call this **multiple linear regression**.

2.1 Multiple Features

2.1.1 Motivation and Extended Notation

Example 2.1.1 (Housing price prediction with multiple features)**.** Suppose we want to predict the selling price of a house. Many attributes influence the price simultaneously:

	Feature	Description	Typical range
x_1	Size (sq. ft.)	Living area	300–2,000
x_2	# Bedrooms	Count	1–5
x_3	# Floors	Count	1–3
x_4	Age (years)	House age	0–80
y	Price ($1,000s)	Target	—

Each row of the training set is one house; each column (except y) is a feature.

We extend our notation as follows.

Definition 2.1.1 (Notation for multiple linear regression)**.** Let the training set contain m examples, each described by n features.

- n — total number of features.
- x_j — the j-th feature $(j = 1, \ldots, n)$.
- $\vec{x}^{(i)} = \left[x_1^{(i)}, x_2^{(i)}, \ldots, x_n^{(i)}\right]^\top$ — the column vector of all feature values for example i.
- $x_j^{(i)}$ — the value of feature j in the i-th example.
- An overhead arrow ($\vec{\cdot}$) denotes a vector; bold font $\mathbf{M}$ denotes a matrix.

Remark 2.1.1. In linear algebra, vector indexing conventionally starts at 1 (i.e. $w_1, w_2, \ldots$), whereas Python and NumPy use zero-based indexing (`w[0]`, `w[1]`, …). This distinction

is important when translating mathematical formulae into code. In particular, `range(n)` iterates from 0 to $n-1$, not from 1 to n.

2.1.2 The Multiple Linear Regression Model

Scalar Form

The model predicts y as a weighted sum of all n features plus a bias b:

$$f_{\vec{w},b}(\vec{x}) \;=\; w_1x_1 + w_2x_2 + \cdots + w_nx_n + b. \tag{2.1}$$

Each weight $w_j \in \mathbb{R}$ measures the *marginal effect* of feature x_j: a one-unit increase in x_j changes the predicted output by w_j, holding all other features fixed. The bias b is the predicted output when all features equal zero; it functions as a *baseline*.

Vector (Dot-Product) Form

Collecting the weights into a row vector $\vec{w} = [w_1, w_2, \ldots, w_n]$ and the features into a column vector $\vec{x} = [x_1, x_2, \ldots, x_n]^\top$, Eq. (2.1) simplifies to

$$\boxed{f_{\vec{w},b}(\vec{x}) \;=\; \vec{w}\cdot\vec{x} + b \;=\; \sum_{j=1}^{n} w_j\, x_j + b.} \tag{2.2}$$

Example 2.1.2 (Interpreting learned parameters)**.** Suppose gradient descent yields

$$f(\vec{x}) = 0.1\,x_1 + 4\,x_2 - 2\,x_3 - 0.5\,x_4 + 80,$$

with features from Example 2.1 and prices in \$1,000s.

- $w_1 = 0.1$: each additional sq. ft. adds \$100 to the predicted price.
- $w_2 = 4$: each extra bedroom adds \$4,000.
- $w_3 = -2$: each additional floor *reduces* the price by \$2,000 (e.g. reflecting higher maintenance).
- $w_4 = -0.5$: each additional year of age reduces the price by \$500.
- $b = 80$: baseline prediction of \$80,000.

2.1.3 Vectorisation for Computational Efficiency

Why Vectorisation Matters

Computing predictions via a `for` loop is *sequential*: each multiplication is executed one after another. Modern CPUs and GPUs contain specialised hardware — SIMD (Single Instruction, Multiple Data) units — that can perform many floating-point operations *simultaneously*. **Vectorisation** phrases computations as linear-algebra operations that numerical libraries such as **NumPy** dispatch to that hardware automatically. For models with thousands of features, the speedup can reduce wall-clock time from hours to minutes.

Three Coding Approaches Compared

Method	Code snippet	Efficiency
Hardcoded	`f = w[0]*x[0]+w[1]*x[1]+...`	Low — not scalable
`for` loop	`for j in range(n): f += w[j]*x[j]`	Medium — readable but slow
Vectorised	`f = np.dot(w, x) + b`	High — uses parallel hardware

NumPy Implementation

```python
import numpy as np

# 4-feature housing example
w = np.array([0.1, 4.0, -2.0, -0.5])    # weight vector  (n,)
b = 80.0                                  # scalar bias
x = np.array([1500, 3, 2, 10])           # one training example (n,)

# Vectorised prediction (single BLAS call)
f = np.dot(w, x) + b
print(f"Predicted price: ${f:.1f}k")      # Predicted price: $228.0k
```

Listing 2.1: Computing a prediction with NumPy

Vectorised Gradient Descent Update

Rather than looping over each weight, all updates are performed in a single array operation:

```python
# dw: NumPy array of partial derivatives, shape (n,)
# db: scalar partial derivative for b
alpha = 0.01

w = w - alpha * dw   # updates ALL weights simultaneously
b = b - alpha * db   # scalar bias update
```

Listing 2.2: Vectorised parameter update

2.1.4 Feature Engineering

Raw features are not always the most informative representation. **Feature engineering** is the practice of constructing new features by transforming or combining existing ones, guided by domain knowledge or exploratory analysis.

Example 2.1.3 (Land area as an engineered feature)**.** A housing dataset may contain x_1: lot frontage (width in metres) and x_2: lot depth (metres). We engineer

$$x_3 = x_1 \times x_2 \quad \text{(total land area in m}^2\text{)}.$$

If area is more predictive of price than either dimension individually, the model will assign a large weight w_3. The expanded model is $f(\vec{x}) = w_1x_1 + w_2x_2 + w_3(x_1x_2) + b$.

2.1.5 Polynomial Regression

Feature engineering is the bridge to **polynomial regression**: by including powers or other non-linear transformations of the original features, we fit *curved* relationships while still using the linear regression framework — because the model remains linear in the *parameters*.

Let x denote house size in sq. ft. Three natural candidates are:

$$\text{Quadratic:} \quad f(x) = w_1 x + w_2 x^2 + b, \tag{2.3}$$

$$\text{Cubic:} \quad f(x) = w_1 x + w_2 x^2 + w_3 x^3 + b, \tag{2.4}$$

$$\text{Square root:} \quad f(x) = w_1 x + w_2 \sqrt{x} + b. \tag{2.5}$$

Each model treats $x, x^2, x^3, \sqrt{x}$ as *separate features* fed into the standard linear regression framework.

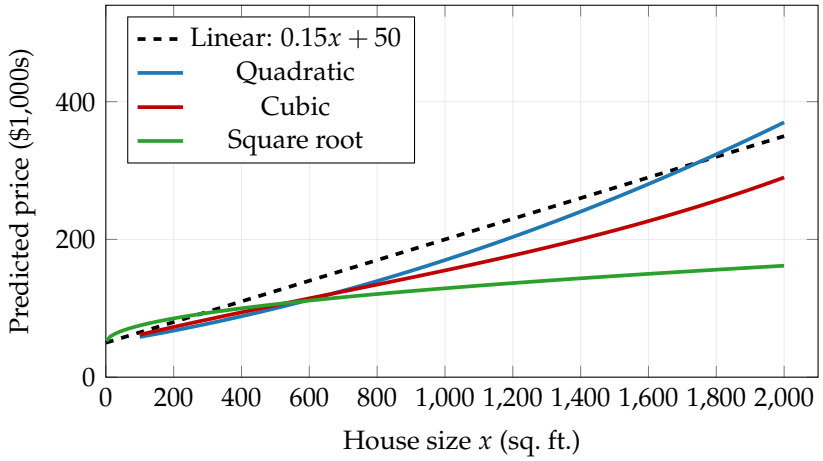


Figure 2.1: Four regression models on hypothetical housing data. The square-root model (green) rises steeply then flattens, the cubic (red) gives a smooth upward trend, and the quadratic (blue) eventually turns downward (possibly unrealistic).

Important: Raising a feature to a high power dramatically expands its numerical range:

$$x \in [1,\ 1{,}000], \quad x^2 \in [1,\ 10^6], \quad x^3 \in [1,\ 10^9].$$

Without feature scaling, one polynomial term would dominate the gradient update. The scaling techniques in Section 2.2.2 must be applied *before* running gradient descent on polynomial features.

2.2 Gradient Descent in Practice

2.2.1 Cost Function and Update Rules for Multiple Features

With m training examples and n features, the cost function generalises naturally:

$$J(\vec{w}, b) = \frac{1}{2m} \sum_{i=1}^{m} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)} \right)^2. \tag{2.6}$$

Gradient descent updates all n weights plus the bias *simultaneously*:

$$w_j \leftarrow w_j - \alpha \frac{\partial J}{\partial w_j}, \quad j = 1, \ldots, n, \tag{2.7}$$

$$b \leftarrow b - \alpha \frac{\partial J}{\partial b}, \tag{2.8}$$

where the partial derivatives are:

$$\frac{\partial J}{\partial w_j} = \frac{1}{m} \sum_{i=1}^{m} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)} \right) x_j^{(i)}, \tag{2.9}$$

$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)} \right). \tag{2.10}$$

Note that Eq. (2.9) differs from its single-feature analogue only by the extra factor $x_j^{(i)}$; the bias update (2.10) is identical in both settings.

Vectorised Implementation

```python
import numpy as np

def compute_gradient(X, y, w, b):
    """
    X  : feature matrix (m, n)
    y  : target vector  (m,)
    w  : weight vector  (n,)
    b  : bias (scalar)
    Returns dw (n,), db (scalar)
    """
    m     = X.shape[0]
    y_hat = X @ w + b          # (m,)
    err   = y_hat - y          # (m,)
    dw    = (X.T @ err) / m    # (n,)
    db    = err.mean()         # scalar
    return dw, db

def gradient_descent(X, y, w, b, alpha, n_iter):
    for _ in range(n_iter):
        dw, db = compute_gradient(X, y, w, b)
        w = w - alpha * dw
        b = b - alpha * db
    return w, b
```

Listing 2.3: Vectorised gradient descent for multiple linear regression

Gradient Descent vs. the Normal Equation

An alternative closed-form solution, the **Normal Equation**, finds the optimal parameters in a single linear algebra step:

$$\vec{w}^* = \left(X^\top X \right)^{-1} X^\top \mathbf{y}.$$

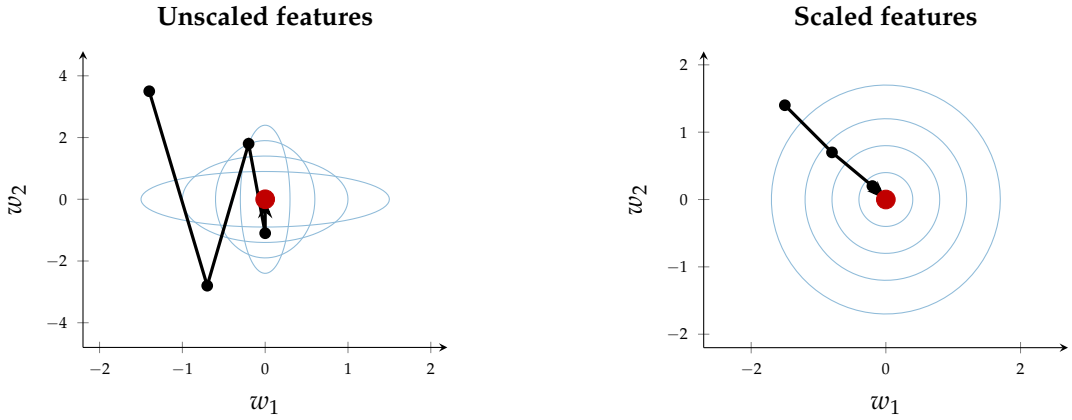
Property	Gradient Descent	Normal Equation
Iterations	Many (iterative)	None (one-shot)
Learning rate	Must be tuned	Not required
Complexity	$O(k \cdot mn)$	$O(n^3)$ for matrix inversion
Scalability	Excellent ($n > 10^6$ feasible)	Slow for $n > 10{,}000$
Applicability	Any ML algorithm	Linear regression only

For large-scale problems, gradient descent (and its stochastic or mini-batch variants) is the standard choice.

2.2.2 Feature Scaling

Motivation

When features have very different numerical ranges, the cost function takes on an elongated, elliptical shape in parameter space (Figure 2.2). Gradient descent then oscillates along the steep walls of these ellipses rather than heading directly toward the minimum, leading to slow convergence. Scaling features to comparable ranges makes the cost function nearly *spherical* and gradient descent converges in far fewer iterations.



(a) Oscillating, slow convergence

(b) Direct, rapid convergence

Figure 2.2: Contour plots of $J(w_1, w_2)$. Without scaling (left), elongated ellipses cause a slow, oscillating gradient path. With scaling (right), the contours are circular and gradient descent converges directly. Red dot marks the global minimum.

Feature Magnitude vs. Parameter Magnitude

There is an **inverse relationship** between a feature's range and the magnitude of its learned weight:

- A feature with a *large* range (e.g. size $\approx$ 1,000 sq. ft.) needs only a *small* weight ($w_1 \approx 0.1$) to produce a reasonable contribution to the prediction.
- A feature with a *small* range (e.g. bedrooms $\approx$ 1–5) needs a *large* weight ($w_2 \approx 50$) to have comparable influence.

Unequal weight magnitudes lead to cost-function contours with very different curvature

along each axis, causing the oscillatory behaviour seen above.

Scaling Techniques

A. Division by Maximum

$$x_j^{\text{scaled}} = \frac{x_j}{\max(x_j)}. \tag{2.11}$$

Maps all values to $[0, 1]$ (assuming non-negative features). Simple but sensitive to outliers.

B. Mean Normalisation

$$x_j \leftarrow \frac{x_j - \mu_j}{\max(x_j) - \min(x_j)}, \tag{2.12}$$

where μ_j is the feature mean. Results fall approximately in $[-1, +1]$.

C. Z-Score Normalisation (Standardisation)

$$x_j \leftarrow \frac{x_j - \mu_j}{\sigma_j}, \tag{2.13}$$

where σ_j is the standard deviation. After this transformation each feature has *zero mean* and *unit variance*. This is the most widely used method and is especially effective when features follow approximately Gaussian distributions.

Example 2.2.1 (Z-score normalisation in numbers)**.** Suppose x_1 (house size) has $\mu_1 = 1{,}000$ sq. ft. and $\sigma_1 = 500$.

House	Raw x_1	Mean-normalised	Z-score
A	300	$\frac{300-1000}{1700} \approx -0.41$	$\frac{300-1000}{500} = -1.40$
B	1000	$\frac{1000-1000}{1700} = 0.00$	$\frac{1000-1000}{500} = 0.00$
C	2000	$\frac{2000-1000}{1700} \approx +0.59$	$\frac{2000-1000}{500} = +2.00$

Practical Guidelines

The target after scaling is approximately $-1 \leq x_j \leq 1$ for each feature.

Feature range (before scaling)	**Rescale?**	**Reason**
$[-3, +3]$ or narrower	Usually not	Acceptable range
$[-0.3, +0.3]$ or narrower	Yes	Too small; gradients negligible
$[-100, +100]$ or wider	Yes	Dominates gradient update
$[0, 0.001]$ or very compressed	Yes	Creates near-zero steps

Remark 2.2.1. Features need not be *perfectly* comparable — merely *roughly* in the same ballpark. When in doubt, always scale: it virtually never hurts performance and almost always accelerates convergence.

Implementation with NumPy and Scikit-learn

```python
import numpy as np
from sklearn.preprocessing import StandardScaler

# --- Manual (NumPy) ---
mu    = X_train.mean(axis=0)       # compute on TRAINING set only
sigma = X_train.std(axis=0)
X_train_sc = (X_train - mu) / sigma
X_test_sc  = (X_test  - mu) / sigma    # apply SAME stats to test set

# --- Scikit-learn ---
scaler = StandardScaler()
X_train_sc = scaler.fit_transform(X_train)    # fit + transform
X_test_sc  = scaler.transform(X_test)         # transform only (no re-
    fit!)
```

Listing 2.4: Z-score normalisation in NumPy and Scikit-learn

Remark 2.2.2. **Always** compute μ_j and σ_j from the *training set only*, then apply those same constants to the validation and test sets. Fitting the scaler on the test set constitutes **data leakage** and leads to overly optimistic performance estimates.

2.2.3 Checking for Convergence

The Learning Curve

A **learning curve** plots the training cost $J(\vec{w}, b)$ against the number of gradient descent iterations. It is the primary diagnostic tool for verifying that training is proceeding correctly.

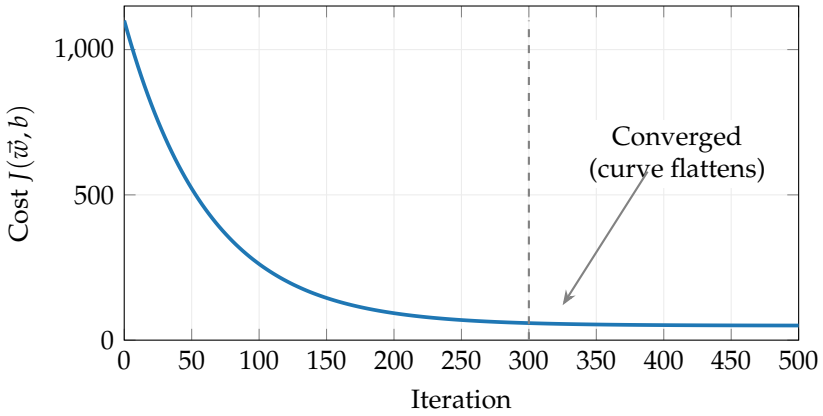


Figure 2.3: A typical learning curve. The cost decreases rapidly at first, then levels off as the algorithm converges. A curve that increases or oscillates signals a learning rate that is too large or a bug in the code.

Key observations:

- A correct implementation always produces a *monotonically decreasing* curve.
- The number of iterations needed varies enormously by problem — from as few as 30 to over 100,000.
- When the curve flattens, gradient descent has approximately converged.

- An *increasing* or *oscillating* curve is a red flag: either α is too large, or there is a bug (e.g. using $+\alpha$ instead of $-\alpha$).

Automatic Convergence Test

Declare convergence when the decrease in cost over one iteration is smaller than a threshold ε (e.g. $\varepsilon = 10^{-3}$):

$$\left|J^{(\text{iter})} - J^{(\text{iter}-1)}\right| \leq \varepsilon.$$

Choosing a reliable ε is difficult in practice; visual inspection of the learning curve is often more informative.

2.2.4 Choosing the Learning Rate

Diagnosing a Poor Learning Rate

- **Monotonically increasing cost**: α is almost certainly too large — the update step overshoots the minimum.
- **Oscillating cost**: α is too large; parameters bounce from one side of the minimum to the other.
- **Very slow decrease**: α is too small; convergence will take an impractical number of iterations.
- **Cost increases even with tiny** α: there is likely a *code bug* — the most common error is using $+\alpha$ in the update rule.

Debugging Strategy

Set α to an extremely small value (e.g. $\alpha = 10^{-10}$) and confirm that J decreases monotonically. If it does not, the bug lies in the gradient computation, not the learning rate.

Grid Search for α

Test a logarithmically spaced range, roughly tripling each candidate:

$$0.001 \;\rightarrow\; 0.003 \;\rightarrow\; 0.01 \;\rightarrow\; 0.03 \;\rightarrow\; 0.1 \;\rightarrow\; 0.3 \;\rightarrow\; 1.$$

For each candidate, run a small number of iterations and plot the learning curve. Select the **largest** value of α that still produces a consistently decreasing curve.

Chapter Summary

- **Multiple linear regression** models the target as $f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$, a weighted sum of n features plus a bias.
- Each weight w_j captures the marginal effect of feature x_j, holding others fixed. Positive w_j indicates a direct relationship; negative indicates an inverse relationship.
- **Vectorisation** (via `np.dot`) replaces slow sequential loops with parallel hardware instructions, yielding dramatic speedups.
- **Feature engineering** (e.g. products, ratios) and **polynomial regression** (powers of x) can reveal more predictive representations while preserving the linear regression

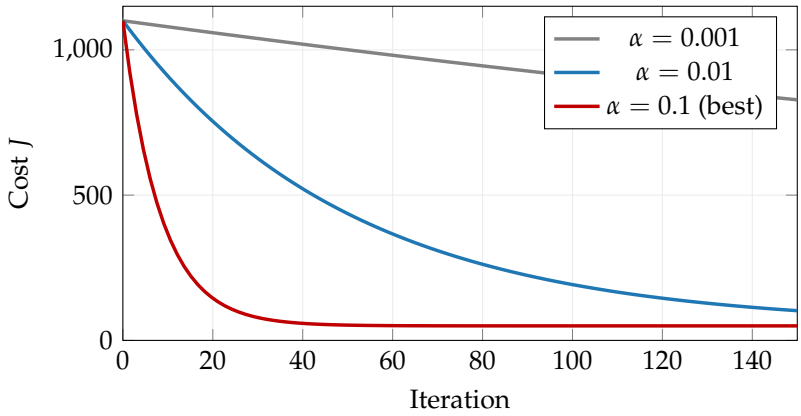


Figure 2.4: Learning curves for three candidate learning rates. The largest stable value ($\alpha = 0.1$) converges the fastest.

framework.

- **Feature scaling** (especially z-score normalisation) maps features to comparable ranges, converting elongated cost ellipses into nearly circular contours, greatly accelerating gradient descent.
- The **learning curve** (cost vs. iterations) is the key diagnostic: a correct implementation yields a monotonically decreasing, eventually flat curve.
- The **learning rate** α should be the largest value that produces stable, monotonic decrease; test candidates on a logarithmic grid.
- The **Normal Equation** offers a closed-form alternative but is computationally impractical for $n \gtrsim 10{,}000$.

Chapter 3

Classification

In the previous two chapters we studied *linear regression*, whose goal was to predict a continuous numerical output such as a house price or a temperature. We now make a fundamental shift: instead of predicting a number along a continuous spectrum, we wish to assign each input to one of a *finite set of categories*. This task is called **classification**.

Classification problems arise everywhere in applied machine learning:

- **Spam detection**: is an incoming email spam or legitimate?
- **Fraud detection**: is a credit-card transaction fraudulent?
- **Medical diagnosis**: is a tumour malignant or benign?
- **Image recognition**: which digit (0–9) appears in this image?
- **Sentiment analysis**: is a product review positive or negative?

The simplest and most common form is **binary classification**, where the output y takes exactly two values. By convention we encode these as:

$$y \in \{0, 1\},$$

where $y = 1$ denotes the *positive class* (presence of the condition) and $y = 0$ the *negative class* (absence). This naming carries no moral connotation.

3.1 Classification with Logistic Regression

3.1.1 Why Linear Regression Is Inadequate for Classification

A natural first instinct is to reuse the linear regression model $f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$ for classification by thresholding: predict $\hat{y} = 1$ if $f > 0.5$, else $\hat{y} = 0$. While this can work in simple situations, it suffers from two fundamental problems.

Problem 1: The outlier effect. Suppose we have a well-separated dataset of benign ($y = 0$) and malignant ($y = 1$) tumours plotted against tumour size. A linear model may yield a reasonable boundary at some threshold x^*. Now add a single extreme outlier — a very large tumour that is still malignant. The least-squares line is pulled toward the outlier, shifting the boundary x^* and misclassifying previously correct points. Linear regression is globally sensitive to every training point; one aberrant observation can corrupt the boundary for the rest.

Problem 2: Unbounded output range. Linear regression produces outputs in $(-\infty, +\infty)$. For a yes/no problem, predicting values such as 1.8 or -0.4 is awkward: they cannot be interpreted as probabilities. We need a model whose output is *guaranteed* to lie in $[0, 1]$.

3.1.2 The Sigmoid (Logistic) Function

The resolution is to *squash* the linear output through a function that maps $\mathbb{R}$ onto $(0, 1)$. The standard choice is the **sigmoid function**:

Definition 3.1.1 (Sigmoid Function)**.**

$$g(z) = \frac{1}{1 + e^{-z}}, \qquad z \in \mathbb{R}.$$

Key properties.

- As $z \to +\infty$: $e^{-z} \to 0$, so $g(z) \to 1$.
- As $z \to -\infty$: $e^{-z} \to \infty$, so $g(z) \to 0$.
- At $z = 0$: $g(0) = \frac{1}{2}$.
- g is smooth, strictly increasing, and S-shaped (sigmoidal).
- Derivative: $g'(z) = g(z)\big(1 - g(z)\big)$.

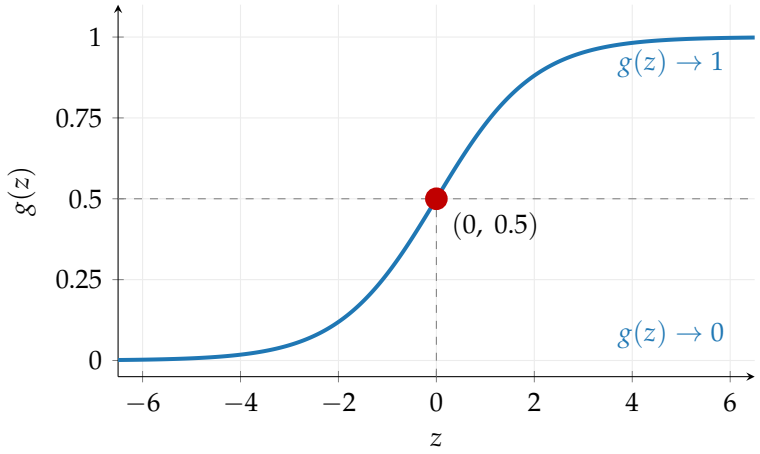


Figure 3.1: The sigmoid (logistic) function. It maps every real number to $(0, 1)$ and equals exactly 0.5 at $z = 0$.

3.1.3 The Logistic Regression Model

Logistic regression is built in two stages.

Stage 1 — Linear combination. Compute a weighted sum:

$$z = \vec{w} \cdot \vec{x} + b.$$

Stage 2 — Sigmoid activation. Pass z through the sigmoid:

$$f_{\vec{w},b}(\vec{x}) = g(z) = g(\vec{w} \cdot \vec{x} + b) = \frac{1}{1 + e^{-(\vec{w} \cdot \vec{x} + b)}}.$$

The output $f_{\vec{w},b}(\vec{x})$ is interpreted as a *probability*:

$$f_{\vec{w},b}(\vec{x}) = P(y = 1 \mid \vec{x};\ \vec{w}, b).$$

This is the estimated probability that the label equals 1 given the input $\vec{x}$ and parameters $(\vec{w}, b)$. The complementary probability is $P(y = 0 \mid \vec{x}) = 1 - f_{\vec{w},b}(\vec{x})$.

Example 3.1.1 (Tumour classification)**.** Let $w = 2$, $b = -5$, $x = 3$ cm (tumour size). Then

$$z = 2 \cdot 3 - 5 = 1, \qquad g(1) = \frac{1}{1 + e^{-1}} \approx 0.731.$$

The model reports a 73.1% probability that the tumour is malignant.

3.1.4 From Probability to Class Prediction

To obtain a hard class label we introduce a **decision threshold** τ (typically $\tau = 0.5$):

$$\hat{y} = \begin{cases} 1 & \text{if } f_{\vec{w},b}(\vec{x}) \geq \tau, \\ 0 & \text{otherwise.} \end{cases}$$

Because $g(z) \geq 0.5 \iff z \geq 0$, the prediction rule is equivalent to:

$$\hat{y} = 1 \iff \vec{w} \cdot \vec{x} + b \geq 0.$$

3.1.5 Decision Boundaries

The **decision boundary** is the set of inputs $\vec{x}$ for which $\vec{w} \cdot \vec{x} + b = 0$ (i.e. where $f = 0.5$, the model is exactly neutral).

Linear boundaries. With two features x_1, x_2 and parameters w_1, w_2, b, the boundary is the straight line $w_1 x_1 + w_2 x_2 + b = 0$. *Example:* $w_1 = 1$, $w_2 = 1$, $b = -3$ gives boundary $x_1 + x_2 = 3$. Points with $x_1 + x_2 > 3$ are classified as $y = 1$.

Non-linear boundaries. By augmenting the feature vector with polynomial terms, the boundary in the *original* feature space can be non-linear. With features (x_1^2, x_2^2) and $w_1 = w_2 = 1$, $b = -1$, the boundary becomes $x_1^2 + x_2^2 = 1$, a unit circle.

3.2 Cost Function for Logistic Regression

3.2.1 Why Squared Error Fails

Reusing the MSE cost from linear regression with the non-linear sigmoid output produces a *non-convex* cost surface with multiple local minima and flat plateaus. Gradient descent may converge to a sub-optimal solution. We need a cost function that (a) preserves convexity and (b) strongly penalises confident wrong predictions.

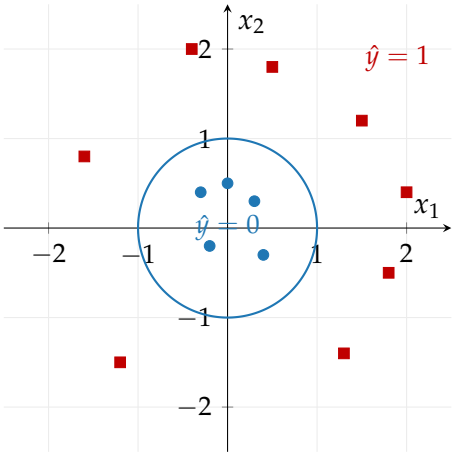


Figure 3.2: Circular decision boundary arising from polynomial features (x_1^2, x_2^2) with $w_1 = w_2 = 1, b = -1$.

3.2.2 Loss vs. Cost: A Key Distinction

Definition 3.2.1 (Loss and Cost)**.** The **loss** $L(f, y)$ measures the penalty on a *single* training example: the cost of predicting f when the true label is y. The **cost** J is the *average loss* over all m training examples:

$$J(\vec{w}, b) = \frac{1}{m} \sum_{i=1}^{m} L\big(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}\big).$$

3.2.3 The Logistic (Cross-Entropy) Loss

We define the loss piecewise:

$$\text{If } y = 1: \quad L = -\log\big(f\big), \tag{3.1}$$

$$\text{If } y = 0: \quad L = -\log\big(1 - f\big). \tag{3.2}$$

Interpretation for $y = 1$ (Eq. 3.1).

- When $f \approx 1$ (correctly confident): $-\log(1) = 0$ — no penalty.
- When $f \to 0$ (confidently wrong): $-\log(f) \to +\infty$ — catastrophic penalty.

Interpretation for $y = 0$ (Eq. 3.2).

- When $f \approx 0$ (correct): $-\log(1) = 0$ — no penalty.
- When $f \to 1$ (confidently wrong): $-\log(0) \to +\infty$ — catastrophic penalty.

3.2.4 The Unified Loss Formula

The piecewise definition collapses elegantly into a single expression:

$$\boxed{L(f, y) = -y \log(f) - (1 - y) \log(1 - f).} \tag{3.3}$$

Verification.

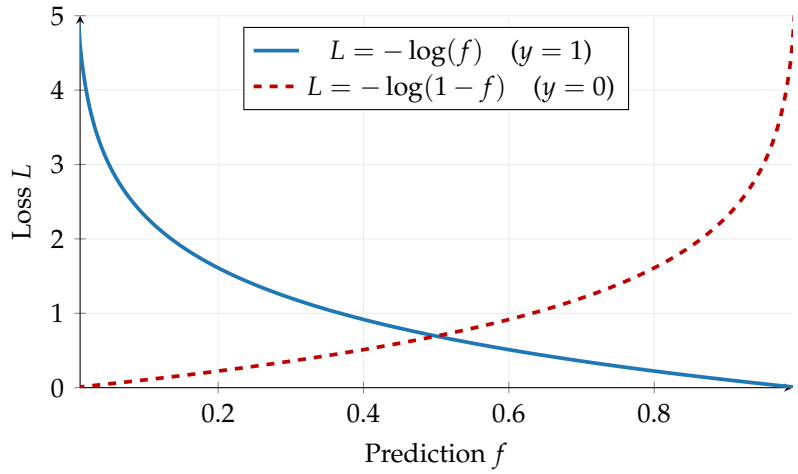


Figure 3.3: Logistic loss functions for $y = 1$ (solid, blue) and $y = 0$ (dashed, red). Both approach $+\infty$ when the prediction is maximally wrong.

- If $y = 1$: $(1 - y) = 0$ annihilates the second term $\Rightarrow L = -\log(f)$.✓
- If $y = 0$: $y = 0$ annihilates the first term $\Rightarrow L = -\log(1 - f)$.✓

3.2.5 The Full Logistic Regression Cost Function

Averaging the unified loss over all m training examples:

$$\boxed{J(\vec{w}, b) = -\frac{1}{m} \sum_{i=1}^{m} \Big[y^{(i)} \log\big(f_{\vec{w},b}(\vec{x}^{(i)})\big) + (1 - y^{(i)}) \log\big(1 - f_{\vec{w},b}(\vec{x}^{(i)})\big) \Big].} \tag{3.4}$$

This is the **cross-entropy loss** or **log-loss**. It has two complementary justifications:

1. **Statistical (MLE).** Eq. (3.4) is exactly the negative log-likelihood of the training data under a Bernoulli model with success probability $f^{(i)}$. Minimising cross-entropy is equivalent to Maximum Likelihood Estimation — a principled statistical objective.
2. **Convexity.** Because the sigmoid is log-concave, this cost function is *convex* in $(\vec{w}, b)$. The cost surface has a *single global minimum* and no local minima, so gradient descent is guaranteed to converge to the optimal solution regardless of initialisation.

3.3 Gradient Descent for Logistic Regression

3.3.1 The Training Objective

We seek $(\vec{w}^*, b^*) = \arg\min_{\vec{w},b} \; J(\vec{w}, b)$. Once found, the trained model estimates $P(y = 1 \mid \vec{x};\ \vec{w}^*, b^*)$ for any new input $\vec{x}$.

3.3.2 The Update Rules

Gradient descent applies simultaneously:

$$w_j \leftarrow w_j - \alpha \frac{\partial J}{\partial w_j}, \quad j = 1, \ldots, n, \tag{3.5}$$

$$b \leftarrow b - \alpha \frac{\partial J}{\partial b}. \tag{3.6}$$

Differentiating Eq. (3.4) (using the chain rule and the identity $g'(z) = g(z)(1 - g(z))$):

$$\frac{\partial J}{\partial w_j} = \frac{1}{m} \sum_{i=1}^{m} \big(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\big)\, x_j^{(i)}, \tag{3.7}$$

$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} \big(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\big). \tag{3.8}$$

Remark 3.3.1. The gradient expressions (3.7)–(3.8) are *structurally identical* to those of linear regression. The critical difference lies in the definition of f: in linear regression $f = \vec{w} \cdot \vec{x} + b$; in logistic regression $f = g(\vec{w} \cdot \vec{x} + b)$. Same form, different meaning.

Derivation sketch for $\partial J / \partial w_j$. Let $f^{(i)} = g(z^{(i)})$, $z^{(i)} = \vec{w} \cdot \vec{x}^{(i)} + b$. The single-example loss is $\ell^{(i)} = -y^{(i)} \log f^{(i)} - (1 - y^{(i)}) \log(1 - f^{(i)})$. Using $\partial f^{(i)} / \partial w_j = f^{(i)}(1 - f^{(i)}) x_j^{(i)}$:

$$\frac{\partial \ell^{(i)}}{\partial w_j} = \Big(-\frac{y^{(i)}}{f^{(i)}} + \frac{1 - y^{(i)}}{1 - f^{(i)}}\Big) f^{(i)}(1 - f^{(i)}) x_j^{(i)} = \big(f^{(i)} - y^{(i)}\big) x_j^{(i)}.$$

Averaging over all m examples gives Eq. (3.7). □

3.3.3 Practical Considerations

Choosing the learning rate α. Monitor the learning curve (cost vs. iteration). A well-chosen α produces a monotonically decreasing, eventually flat curve.

Feature scaling. When features have vastly different magnitudes, the cost surface becomes an elongated ellipse. Standardising features (zero mean, unit variance) transforms the contours to circles, allowing much faster convergence.

Vectorisation. For large datasets, express the gradient update as matrix–vector operations (NumPy) to exploit parallelism.

Convergence criterion. Halt when the relative change in J between successive iterations falls below a tolerance ε:

$$\frac{|J^{(t)} - J^{(t-1)}|}{|J^{(t-1)}| + \varepsilon_0} < \varepsilon.$$

3.4 The Problem of Overfitting

3.4.1 The Bias–Variance Trade-off

Every supervised learning algorithm navigates a fundamental tension: a model that is too simple cannot capture the true structure (**underfitting**, or *high bias*), while a model that is too complex memorises the training data and fails to generalise (**overfitting**, or *high variance*).

Underfitting (High Bias). An underfitting model has a strong preconception about the form of the relationship. Fitting a straight line to clearly curved data forces the model to ignore obvious patterns. Such a model performs poorly even on the training set.

Overfitting (High Variance). An overfitting model is excessively complex — perhaps a degree-15 polynomial fitted to 20 data points. It interpolates the training data perfectly (near-zero training error) but its oscillatory curve is driven by noise rather than the true signal. Small changes in the training set produce drastically different models.

Good generalisation. The ideal model captures the true underlying pattern while ignoring random noise. It achieves low error on both training and unseen test data.

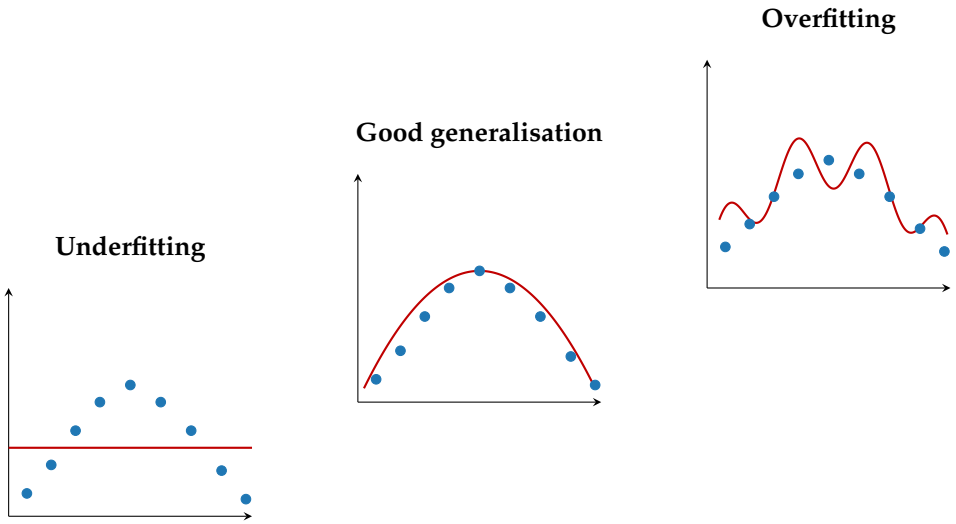


Figure 3.4: Three models fitted to the same data. *Left*: underfitting (too simple). *Centre*: good generalisation. *Right*: overfitting (too complex, fitting the noise).

3.4.2 Remedies for Overfitting

Strategy 1: Collect More Training Data

The most direct remedy. With more data, any fixed-complexity model is less able to memorise individual points and is forced to learn the true underlying pattern. Unfortunately, collecting more data is not always feasible — it may be expensive, time-consuming, or impossible for rare events.

Strategy 2: Feature Selection

If the number of features n is large relative to m, overfitting is likely. Reducing the feature set — keeping only the most informative features — reduces the model's capacity to overfit. Feature selection can be done manually (guided by domain knowledge) or automatically (using statistical tests, embedded methods, or regularisation paths).

Strategy 3: Regularisation

Regularisation is generally the preferred remedy because it allows us to retain *all* features while preventing any single parameter from dominating the model.

Core idea. Large weight parameters are what allow a model to oscillate wildly. By *penalising* large weights in the cost function, the optimiser is forced to keep them small, producing a smoother, more parsimonious model.

Regularised cost function (L_2 / Ridge).

$$J_{\text{reg}}(\vec{w}, b) = J(\vec{w}, b) + \frac{\lambda}{2m} \sum_{j=1}^{n} w_j^2, \tag{3.9}$$

where $\lambda \geq 0$ is the **regularisation parameter**. Note that b is conventionally *not* regularised.

The role of λ.

- $\lambda = 0$: no regularisation; the model may overfit.
- $\lambda \to \infty$: all weights forced to zero; model underfits (predicts constant $f = \sigma(b)$).
- λ "just right": balance between fitting data and model simplicity, achieving good generalisation.

3.4.3 Regularised Linear Regression

Gradient descent updates.

$$w_j \leftarrow w_j - \alpha \Big[\frac{1}{m} \sum_{i=1}^{m} \big(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)} \big) x_j^{(i)} + \frac{\lambda}{m} w_j \Big], \tag{3.10}$$

$$b \leftarrow b - \frac{\alpha}{m} \sum_{i=1}^{m} \big(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)} \big). \tag{3.11}$$

Weight decay interpretation. Rearranging (3.10):

$$w_j \leftarrow \underbrace{\Big(1 - \frac{\alpha\lambda}{m}\Big)}_{\text{weight decay factor}} w_j - \frac{\alpha}{m} \sum_{i=1}^{m} \big(f^{(i)} - y^{(i)} \big) x_j^{(i)}.$$

Since $\alpha\lambda/m$ is a small positive number (e.g. 0.0002), the weight decay factor is slightly less than 1. At every step, each weight is *gently shrunk* before the gradient update, creating a constant restoring force toward zero.

3.4.4 Regularised Logistic Regression

The same approach applies directly:

$$J(\vec{w}, b) = -\frac{1}{m}\sum_{i=1}^{m}\Big[y^{(i)}\log f^{(i)} + (1-y^{(i)})\log(1-f^{(i)})\Big] + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2, \tag{3.12}$$

with gradient descent updates identical in form to (3.10)–(3.11) but with the sigmoid f.

Summary table.

Situation	J_{train}	J_{test}	**Remedy**
Underfitting (High Bias)	High	High	More features; reduce λ
Good generalisation	Low	Low	—
Overfitting (High Variance)	Low	High	More data; reduce features; increase λ

Chapter Summary

- **Logistic regression** uses the sigmoid function to map a linear combination of features onto $(0, 1)$, yielding a probability estimate $P(y = 1 \mid \vec{x})$.
- The decision boundary $\vec{w} \cdot \vec{x} + b = 0$ is linear; polynomial features extend it to non-linear shapes (circles, ellipses, etc.).
- The **cross-entropy cost** is derived from MLE and is convex, guaranteeing a unique global minimum. Squared error would produce a non-convex cost with local minima.
- The gradient formulas for logistic regression look identical to those of linear regression, but f is the sigmoid in the logistic case.
- **Overfitting** occurs when the model is too complex relative to the training data. The three main remedies are more data, feature selection, and L_2 **regularisation** (controlled by λ).
- Regularisation adds a weight-penalty term to the cost, shrinking weights at every gradient step ("weight decay"), smoothing the decision boundary.

Chapter 4

Neural Networks

4.1 Neural Networks Intuition

4.1.1 Biological Inspiration and Engineering Reality

The original motivation behind **Artificial Neural Networks (ANNs)** — often called simply *neural networks* or *deep learning* models — was to create software that mimics the learning processes of the human brain. While modern engineering has largely moved beyond strict biological analogies, the foundational vocabulary still draws on neuroscience.

A **biological neuron** processes information in three stages:

1. **Dendrites** receive incoming electrical impulses from other neurons.
2. The **cell body (soma)** aggregates and processes these signals.
3. The **axon** transmits the resulting output to downstream neurons.

An **artificial neuron** is a mathematical abstraction of this process. It takes numerical inputs, computes a weighted sum, applies a non-linear function, and emits an *activation value*.

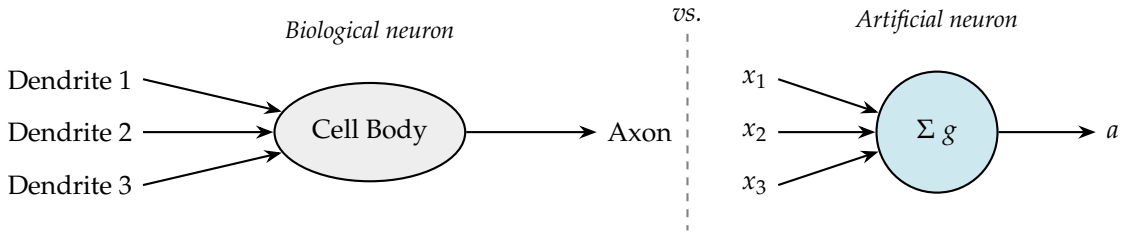


Figure 4.1: Structural analogy between a biological neuron and an artificial neuron. The artificial neuron computes a weighted sum (Σ) and applies a non-linear function (g) to produce the activation a.

Remark 4.1.1. Despite the name, modern neural networks have very little to do with how the brain actually works. We still understand remarkably little about neuroscience. Today's AI systems are driven primarily by *engineering principles* and large-scale empirical results rather than by neurological simulation.

4.1.2 A Brief History and the Data-Compute Story

The development of neural networks has passed through alternating periods of enthusiasm and neglect:

Era	Status	Key developments
1950s	Inception	Brain-inspired computing first proposed
1980s–early 1990s	First wave	Handwritten digit recognition (zip codes)
Late 1990s	Decline	Outperformed by SVMs and other ML methods
2005–present	Resurgence	Rebranded as "Deep Learning"; breakthroughs

The key milestones in the modern era include: speech recognition (early 2010s), ImageNet 2012 (landmark computer vision result), NLP with Transformers, and today's large language models.

Why now? Traditional algorithms exhibit a characteristic performance ceiling; beyond a certain amount of training data, adding more examples yields diminishing returns. Large neural networks, by contrast, continue to improve as datasets grow. Two enabling factors: the explosion of digital data from the internet, and GPU hardware providing the massive parallel computation needed.

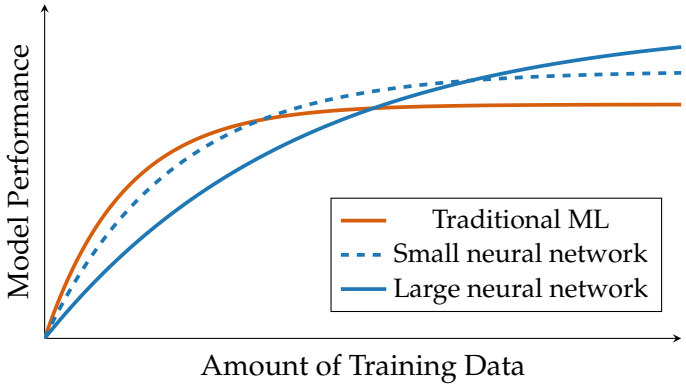


Figure 4.2: Schematic performance versus training data size. Large neural networks continue to benefit from additional data, whereas classical methods plateau.

4.2 Neural Network Model

4.2.1 The Single Neuron

A single artificial neuron applies a **sigmoid activation function**:

$$a = g(z) = \frac{1}{1 + e^{-z}}, \qquad z = wx + b. \tag{4.1}$$

The term *activation* is borrowed from neuroscience: just as a biological neuron fires when stimulated strongly enough, an artificial neuron emits a large output when its weighted input is large.

4.2.2 Layered Architecture

When multiple neurons are combined in a layered structure, the network can capture complex non-linear relationships. A standard feedforward network has:

- **Input layer (Layer 0)**: the raw feature vector $\vec{x} \in \mathbb{R}^n$, also denoted $\vec{a}^{[0]}$.
- **Hidden layers**: one or more intermediate layers. Each neuron in a hidden layer is a logistic-regression unit whose inputs are the activations of the previous layer.
- **Output layer**: produces the final prediction.

The term *hidden* means these layers are not directly observed in the training data — we see only inputs $\vec{x}$ and labels y.

4.2.3 Notation

- Superscript $[l]$ denotes the **layer index**: $[1], [2], \ldots$
- Subscript j denotes the **neuron index** within a layer.
- $a_j^{[l]}$: activation of neuron j in layer l.
- $\vec{w}_j^{[l]} \in \mathbb{R}^{n_{l-1}}$, $b_j^{[l]} \in \mathbb{R}$: weight vector and bias.
- $\vec{a}^{[l]}$: vector of all activations in layer l.
- $\vec{a}^{[0]} = \vec{x}$ by definition.

4.2.4 The General Activation Formula

For any layer l and unit j:

$$\boxed{a_j^{[l]} = g\Big(\vec{w}_j^{[l]} \cdot \vec{a}^{[l-1]} + b_j^{[l]}\Big).} \tag{4.2}$$

This single formula, applied repeatedly across layers and units, describes the entire forward computation of a feedforward neural network.

4.2.5 Layered Computation: Example

Consider a network with architecture $n = 4$ inputs, hidden layers of sizes 3 and 3, and a single output.

Layer 1 (3 neurons, input $\vec{a}^{[0]} = \vec{x}$):

$$\begin{aligned} a_1^{[1]} &= g(\vec{w}_1^{[1]} \cdot \vec{x} + b_1^{[1]}), \\ a_2^{[1]} &= g(\vec{w}_2^{[1]} \cdot \vec{x} + b_2^{[1]}), \\ a_3^{[1]} &= g(\vec{w}_3^{[1]} \cdot \vec{x} + b_3^{[1]}). \end{aligned}$$

Layer output: $\vec{a}^{[1]} = [a_1^{[1]}, a_2^{[1]}, a_3^{[1]}]^\top$.

Layer 2 (output, 1 neuron):

$$a_1^{[2]} = g(\vec{w}_1^{[2]} \cdot \vec{a}^{[1]} + b_1^{[2]}) \in (0, 1).$$

Prediction: $\hat{y} = 1$ if $a_1^{[2]} \geq 0.5$, else $\hat{y} = 0$.

4.2.6 Deep Architectures and Hierarchical Features

Networks with more than one hidden layer are called **deep networks**, and the field studying them is **deep learning**. In computer vision, visualisation studies reveal a remarkable hierarchy of learned features:

Layer	Features detected	Description
1st hidden	Edges, oriented lines	Short segments at various angles
2nd hidden	Object parts	Eyes, corners, wheels
3rd hidden	Coarse shapes	Full face or car outlines
Output	Category/identity	Final classification probability

Crucially, **no human defines these features** — the network discovers them automatically by minimising prediction error.

4.3 TensorFlow Implementation

4.3.1 Data Representation

TensorFlow processes data in **two-dimensional matrices** (tensors). Always use 2D arrays when passing data to layers (shape: samples $\times$ features).

4.3.2 The Dense Layer and Sequential API

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Define architecture
model = Sequential([
    Dense(units=3, activation='sigmoid'),    # Hidden layer: 3 neurons
    Dense(units=1, activation='sigmoid')     # Output layer: 1 neuron
])
```

Listing 4.1: Sequential model for coffee roasting classification

4.3.3 The Training Workflow

After defining architecture, training follows three steps:

1. **Compile** — specify loss and optimiser:

```
model.compile(loss='binary_crossentropy', optimizer='adam')
```

2. **Train** — fit the model:

```
model.fit(X_train, y_train, epochs=100)
```

3. **Predict**:

```
predictions = model.predict(X_new)
```

4.3.4 Complete Example: Digit Recognition

For 8×8 pixel handwritten digit images (64 input features):

```
model = Sequential([
    Dense(units=25, activation='sigmoid'),   # Layer 1: 25 units
    Dense(units=15, activation='sigmoid'),   # Layer 2: 15 units
    Dense(units=1,  activation='sigmoid')    # Output:   1 unit
])
```

Listing 4.2: Neural network for handwritten digit recognition

The network compresses the representation from 64 dimensions down to a single probability.

4.4 Forward Propagation in NumPy

Understanding how to implement forward propagation manually is essential for debugging and appreciating the linear algebra underlying every deep learning framework.

```
import numpy as np

def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))

def dense(a_in, W, b, activation=sigmoid):
    """
    a_in : (n_in,)       activations from previous layer
    W    : (n_in, units) weight matrix (each COLUMN = one neuron)
    b    : (units,)      bias vector
    """
    n_units = W.shape[1]
    a_out   = np.zeros(n_units)
    for j in range(n_units):
        w_j       = W[:, j]
        z_j       = np.dot(w_j, a_in) + b[j]
        a_out[j]  = activation(z_j)
    return a_out

def sequential(x, W1, b1, W2, b2):
    """Full forward pass through a 2-layer network."""
    a1  = dense(x,  W1, b1)
    a2  = dense(a1, W2, b2)
    return a2
```

Listing 4.3: Reusable `dense()` layer function

4.4.1 Vectorised Implementation

```python
def dense_vectorised(A_in, W, b, activation=sigmoid):
    """
    A_in : (1, n_in) -- 2D row vector input
    W    : (n_in, units)
    b    : (1, units) or broadcastable
    Returns A_out : (1, units)
    """
    Z     = np.matmul(A_in, W) + b    # (1, units)
    A_out = activation(Z)              # element-wise
    return A_out
```

Listing 4.4: Vectorised dense layer using `np.matmul`

For a batch of m examples, A_{in} has shape $(m \times n_{\text{in}})$ and one `matmul` computes all m predictions simultaneously.

4.5 Speculations on Artificial General Intelligence

Current neural networks are **Artificial Narrow Intelligence (ANI)**: trained for a specific task. **Artificial General Intelligence (AGI)** would be capable of any intellectual task a human can perform and remains an unsolved problem. Credible expert estimates for AGI range from "within the decade" to "never, as currently conceived" — highlighting how poorly we understand both the requirements for general intelligence and the trajectory of AI progress.

Chapter Summary

- A neural network stacks layers of artificial neurons, each computing $a_j^{[l]} = g(\vec{w}_j^{[l]} \cdot \vec{a}^{[l-1]} + b_j^{[l]})$.
- Deep networks automatically learn hierarchical features from data without any human-defined feature engineering.
- TensorFlow's Sequential API defines, compiles, and trains a network in a few lines; the training loop internally uses backpropagation.
- NumPy-level implementation reveals the matrix multiply at the core of every deep learning framework.
- Vectorisation (batch matrix operations) is essential for efficiency.

Chapter 5

Neural Network Training

5.1 A Three-Step Training Framework

Whether fitting a two-parameter logistic regression or a million-parameter deep network, the workflow decomposes into exactly three conceptual steps:

Step	Goal	Logistic Regression	Neural Network (TF)
1	Define $\vec{x} \to \hat{y}$	$f = \sigma(\vec{w} \cdot \vec{x} + b)$	`Sequential([Dense(...)])`
2	Measure error	Logistic loss	`model.compile(loss=...)`
3	Minimise error	Gradient descent	`model.fit(X,y,epochs=...)`

5.1.1 Step 1 — Architecture

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
    Dense(units=25, activation='sigmoid'),  # Hidden layer 1
    Dense(units=15, activation='sigmoid'),  # Hidden layer 2
    Dense(units=1,  activation='sigmoid'),  # Output layer
])
```

Listing 5.1: Binary digit classifier: 3-layer network

TensorFlow initialises all weight matrices and bias vectors automatically.

5.1.2 Step 2 — Compile: Specify the Loss Function

Binary Cross-Entropy Loss

For $(\vec{x}, y)$ with $y \in \{0, 1\}$:

$$L(f, y) = -y \log f - (1 - y) \log(1 - f). \tag{5.1}$$

Cost over all m examples:

$$J(\mathbf{W}, \mathbf{B}) = \frac{1}{m} \sum_{i=1}^{m} L\big(f(\vec{x}^{(i)}), y^{(i)}\big). \tag{5.2}$$

```
from tensorflow.keras.losses import BinaryCrossentropy
```

```
model.compile(loss=BinaryCrossentropy())
```

Mean Squared Error (Regression)

```
from tensorflow.keras.losses import MeanSquaredError
model.compile(loss=MeanSquaredError())
```

5.1.3 Step 3 — Train: Backpropagation

To minimise J, every parameter in every layer is updated iteratively:

$$w_j^{[l]} \leftarrow w_j^{[l]} - \alpha \frac{\partial J}{\partial w_j^{[l]}}, \tag{5.3}$$

$$b_j^{[l]} \leftarrow b_j^{[l]} - \alpha \frac{\partial J}{\partial b_j^{[l]}}. \tag{5.4}$$

Computing these partial derivatives across all layers is **backpropagation** (Section 5.5).

Definition 5.1.1 (Epoch)**.** One **epoch** is a single complete pass of gradient descent over the entire training set.

```
model.fit(X, Y, epochs=100)
```

5.2 Activation Functions

5.2.1 Why Not Just Use Sigmoid Everywhere?

Using sigmoid in every hidden layer causes two problems:

1. **Range constraint.** Sigmoid outputs lie in $(0, 1)$, but hidden-layer activations may represent quantities with unbounded range.
2. **Vanishing gradients.** At large $|z|$, $g'(z) \approx 0$, causing gradients to shrink exponentially during backpropagation through many layers.

5.2.2 Catalogue of Common Activation Functions

Sigmoid

$$g(z) = \frac{1}{1 + e^{-z}}, \quad g(z) \in (0, 1). \tag{5.5}$$

Natural choice for binary classifier output.

ReLU (Rectified Linear Unit)

$$g(z) = \max(0, z), \quad g(z) \in [0, \infty). \tag{5.6}$$

The **default for hidden layers** in modern deep learning. Advantages:

- **Computationally cheap**: a single comparison, no exponentiation.
- **Non-vanishing gradients**: for $z > 0$ the gradient is exactly 1.

Linear (Identity)

$$g(z) = z, \quad g(z) \in (-\infty, +\infty). \tag{5.7}$$

Appropriate *only* in the output layer for regression (any-sign y).

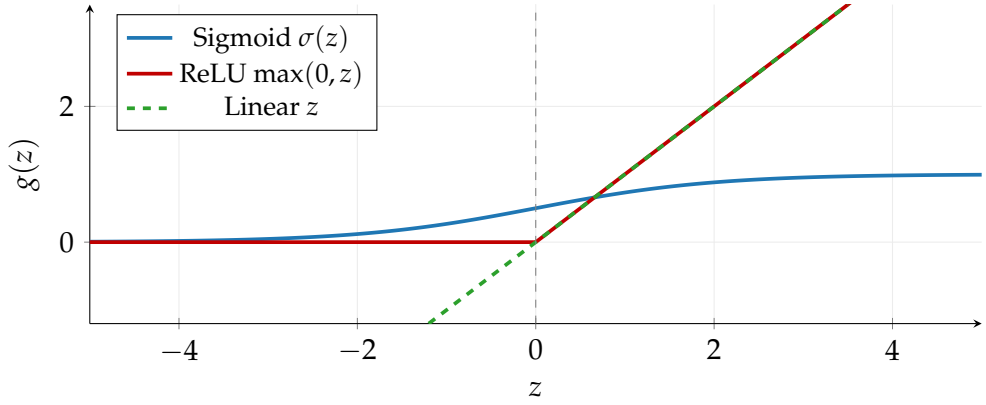


Figure 5.1: Comparison of the three most common activation functions.

5.2.3 Choosing the Right Activation

Output Layer — Determined by Task

Task	Target y	Activation	Range
Binary classification	$y \in \{0, 1\}$	Sigmoid	$(0, 1)$
Regression (any sign)	$y \in \mathbb{R}$	Linear	$(-\infty, +\infty)$
Regression (≥ 0)	$y \geq 0$	ReLU	$[0, +\infty)$

Hidden Layers — ReLU as Default

```
model = Sequential([
    Dense(units=25, activation='relu'),
    Dense(units=15, activation='relu'),
    Dense(units=1,  activation='sigmoid'),
])
```

Listing 5.2: Recommended: ReLU hidden layers with sigmoid output

5.2.4 The Necessity of Non-Linearity

What happens if we use *linear* activations everywhere?

Mathematical proof (one hidden unit).

$$a^{[1]} = w_1 x + b_1,$$
$$a^{[2]} = w_2 a^{[1]} + b_2 = w_2(w_1 x + b_1) + b_2 = (w_2 w_1)x + (w_2 b_1 + b_2) = Wx + B.$$

Setting $W = w_2 w_1$ and $B = w_2 b_1 + b_2$ gives a *linear* model. The hidden layer contributed nothing.

Generalisation: any depth of linear layers is equivalent to a single linear layer. Adding more linear layers provides *zero* additional expressive power. Non-linear activations (ReLU) are what allow deep networks to learn hierarchical, complex features.

Rule of thumb.

- **Never** use linear activations in hidden layers.
- Use **ReLU** for hidden layers.
- Use **linear** in the output only for regression with $y \in \mathbb{R}$.

5.3 Multiclass Classification

5.3.1 Definition and Motivation

Definition 5.3.1 (Multiclass classification)**.** A problem is **multiclass** when $y \in \{1, 2, \ldots, N\}$ for $N > 2$.

Examples: handwritten digit recognition ($N = 10$), medical diagnosis with multiple disease types, part-of-speech tagging.

5.3.2 Softmax Regression

Logistic regression is a special case of *softmax regression* with $N = 2$. For general N, each class j has its own parameter vector $(\vec{w}_j, b_j)$.

Step 1: linear terms.

$$z_j = \vec{w}_j \cdot \vec{x} + b_j, \quad j = 1, \ldots, N. \tag{5.8}$$

Step 2: softmax activation.

$$\boxed{a_j = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}} = P(y = j \mid \vec{x}).} \tag{5.9}$$

Normalisation property.

$$\sum_{j=1}^{N} a_j = 1. \tag{5.10}$$

Remark 5.3.1 (Coupling across units)**.** Unlike sigmoid or ReLU, softmax is **not element-wise**. Computing a_j requires all N values of z simultaneously due to the shared denominator.

5.3.3 Cross-Entropy Loss for Softmax

If the ground-truth label for example i is $y^{(i)} = j$:

$$L = -\log(a_j) \quad \text{if } y = j. \tag{5.11}$$

The total cost: $J = \frac{1}{m} \sum_{i=1}^{m} L(f(\vec{x}^{(i)}), y^{(i)})$.

5.3.4 TensorFlow Implementation

```
from tensorflow.keras.losses import SparseCategoricalCrossentropy

model = Sequential([
    Dense(units=25, activation='relu'),
    Dense(units=15, activation='relu'),
    Dense(units=10, activation='softmax'),    # Softmax output
])
model.compile(loss=SparseCategoricalCrossentropy())
model.fit(X, Y, epochs=100)
```

Listing 5.3: Softmax network for 10-class digit recognition

5.4 Advanced Concepts

5.4.1 Numerical Stability: The `from_logits` Pattern

When TensorFlow evaluates softmax then log inside the cross-entropy loss, it may suffer from floating-point overflow or underflow. The solution: let the final layer output raw **logits** (the z values) and pass `from_logits=True`. TensorFlow then combines the operations into a single numerically stable formula.

Definition 5.4.1 (Logit)**.** A **logit** is the raw, unscaled output z of the final layer before any activation.

```
model = Sequential([
    Dense(units=25, activation='relu'),
    Dense(units=15, activation='relu'),
    Dense(units=10, activation='linear'),    # Output logits, not
        probabilities
])
model.compile(loss=SparseCategoricalCrossentropy(from_logits=True))
model.fit(X, Y, epochs=100)

# Convert logits to probabilities at inference time:
import tensorflow as tf
logits = model.predict(X)
probs  = tf.nn.softmax(logits)
```

Listing 5.4: Numerically stable multiclass model (recommended in production)

5.4.2 Multi-Label Classification

Definition 5.4.2 (Multi-label classification)**.** The model must predict a *vector* of binary labels; multiple labels can be simultaneously active (non-mutually exclusive).

Example 5.4.1. In autonomous driving, a single image may contain a car ($y_1 = 1$), no bus ($y_2 = 0$), and a pedestrian ($y_3 = 1$) simultaneously.

Feature	Multi-class	Multi-label
Output y	Single integer	Vector of N binaries
Classes	Mutually exclusive	Independent
Output activation	Softmax	Sigmoid (per output)
Loss	Sparse categorical CE	Binary CE

```
model = Sequential([
    Dense(units=25, activation='relu'),
    Dense(units=15, activation='relu'),
    Dense(units=3,  activation='sigmoid'),    # 3 independent binary
        outputs
])
model.compile(loss='binary_crossentropy')
```

Listing 5.5: Multi-label classification model

5.4.3 The Adam Optimiser

Standard gradient descent uses a single fixed α for all parameters.

Definition 5.4.3 (Adam). **Adam** (Adaptive Moment Estimation) maintains a *separate, adaptive learning rate* α_j for every parameter w_j.

Intuition:

- If w_j consistently moves in the *same direction*: increase α_j.
- If w_j *oscillates*: decrease α_j.

Adam maintains two running statistics (moments) per parameter: a first moment (exponentially weighted average of gradients) and a second moment (weighted average of squared gradients). Their ratio yields a per-parameter step size.

```
from tensorflow.keras.optimizers import Adam
model.compile(
    optimizer=Adam(learning_rate=1e-3),
    loss=SparseCategoricalCrossentropy(from_logits=True))
model.fit(X, Y, epochs=100)
```

Listing 5.6: Using Adam in TensorFlow

5.5 Backpropagation

5.5.1 The Intuition Behind Derivatives

Backpropagation computes the partial derivative of the cost J with respect to every parameter. These derivatives are used by the optimiser to update parameters.

Definition 5.5.1 (Derivative (informal)). $\frac{\partial J}{\partial w} \approx k$ means: increasing w by ε causes J to change by approximately $k\varepsilon$.

Example: $J(w) = w^2$**.**

w	$J = w^2$	$w + \varepsilon$	ΔJ	$\dfrac{\partial J}{\partial w} = 2w$
3	9	3.001	$\approx 6\varepsilon$	6
2	4	2.001	$\approx 4\varepsilon$	4
-3	9	-2.999	$\approx -6\varepsilon$	-6

The derivative tells the optimiser: (1) which *direction* to move, and (2) how large a *step* is appropriate.

5.5.2 Computation Graphs and the Chain Rule

Definition 5.5.2 (Computation graph)**.** A **computation graph** is a DAG where each node represents an elementary operation and each edge carries a value. Forward propagation traverses it left-to-right; backpropagation right-to-left using the chain rule.

Forward pass example. $a = wx + b$, $d = a - y$, $J = \frac{1}{2}d^2$. With $x = -2$, $y = 2$, $w = 2$, $b = 8$:

$$c = 2(-2) = -4, \quad a = -4 + 8 = 4, \quad d = 4 - 2 = 2, \quad J = 2.$$

Backward pass (chain rule).

$$\frac{\partial J}{\partial d} = d = 2, \quad \frac{\partial J}{\partial a} = \frac{\partial J}{\partial d} \cdot 1 = 2,$$
$$\frac{\partial J}{\partial b} = 2, \quad \frac{\partial J}{\partial c} = 2, \quad \frac{\partial J}{\partial w} = \frac{\partial J}{\partial c} \cdot x = 2 \cdot (-2) = -4.$$

5.5.3 Computational Efficiency

Method	**Complexity**	**Description**
Naive numerical	$O(N \times P)$	Re-run forward pass for each parameter
Backpropagation	$O(N + P)$	Single backward pass, reuse partial derivatives

For a network with one million parameters, the naive approach requires one million forward passes per gradient step — physically intractable. Backpropagation collapses this to a single backward pass of cost comparable to one forward pass. Modern frameworks (TensorFlow, PyTorch) implement **automatic differentiation**: they construct the computation graph dynamically and traverse it in reverse, computing all gradients without any user-supplied formulas.

Chapter Summary

Concept	Key takeaway
3-step training	Specify $\rightarrow$ Compile $\rightarrow$ Fit
Activation choice	ReLU for hidden; task-determined for output
Linear collapse	All-linear networks $\equiv$ linear regression
Multiclass	Softmax output with N neurons
Numerical stability	Use `from_logits=True` in production
Multi-label	N independent sigmoid outputs
Adam	Per-parameter adaptive learning rates
Backpropagation	$O(N+P)$ vs. $O(N \times P)$ naive; chain rule
Autodiff	TF/PyTorch automate all gradient computation

Chapter 6

Advice for Applying Machine Learning

This chapter provides a systematic framework for diagnosing and improving machine learning systems. We cover practical debugging strategies, the bias–variance trade-off, structured development workflows, data-centric AI, and methods for handling skewed datasets.

6.1 Evaluating Model Performance

6.1.1 Unacceptable Prediction Errors: What Next?

Suppose regularised linear regression produces unacceptably large errors. You face a critical decision: *what should you try next?* Choosing wrongly can waste months of effort. Six commonly considered paths exist:

Strategy	Description
Data collection	Gather more training examples.
Feature reduction	Use a smaller feature set.
Feature expansion	Introduce additional informative features.
Polynomial complexity	Add polynomial terms for non-linear patterns.
Decrease λ	Allow closer fitting of training data.
Increase λ	Reduce overfitting by stronger regularisation.

Without guidance, practitioners often choose randomly. **Machine learning diagnostics** replace guessing with evidence-based reasoning.

Definition 6.1.1 (Diagnostic)**.** A **diagnostic** is a formal test that provides insight into what is or is not working within a learning algorithm, guiding targeted improvement.

6.1.2 The Train / Test Split

The dataset is divided into two non-overlapping subsets:

1. **Training set** ($\approx$ 70–80%): used to fit parameters $(\vec{w}, b)$.
2. **Test set** ($\approx$ 20–30%): used to evaluate on unseen data.

Procedure for linear regression.

1. Minimise the regularised cost on the training set.
2. Compute train and test errors *without* regularisation:

$$J_{\text{test}} = \frac{1}{2m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} \left(f_{\vec{w},b}(\vec{x}_{\text{test}}^{(i)}) - y_{\text{test}}^{(i)} \right)^2, \tag{6.1}$$

$$J_{\text{train}} = \frac{1}{2m_{\text{train}}} \sum_{i=1}^{m_{\text{train}}} \left(f_{\vec{w},b}(\vec{x}_{\text{train}}^{(i)}) - y_{\text{train}}^{(i)} \right)^2. \tag{6.2}$$

Remark 6.1.1 (Overfitting diagnostic rule)**.** If J_{train} is low but J_{test} is high, the model has **failed to generalise** (overfitting).

6.1.3 The Three-Way Data Split for Model Selection

A two-way split has a flaw: if we use the test set to select the best model (e.g. the best polynomial degree d), J_{test} is no longer an unbiased estimate — it has been "used up" in the selection decision.

Solution: split data into three disjoint subsets.

Subset	Size	Purpose
Training set	60%	Fit parameters $\vec{w}$, b
Cross-validation (CV)	20%	Select model / tune hyperparameters
Test set	20%	Unbiased final performance estimate

Correct procedure.

1. Train each candidate model on the training set.
2. Evaluate all models on the CV set; choose the one with lowest J_{cv}.
3. Report J_{test} of the *single* chosen model.

6.2 Bias and Variance

6.2.1 Defining Bias and Variance

Definition 6.2.1 (High Bias)**.** A model has **high bias** when it is too simple to capture the underlying patterns. Both J_{train} and J_{cv} are high, and close to each other.

Definition 6.2.2 (High Variance)**.** A model has **high variance** when it fits training noise and fails to generalise. J_{train} is very low but $J_{\text{cv}} \gg J_{\text{train}}$.

Condition	J_{train}	J_{cv}	Relationship
High Bias	High	High	$J_{\text{cv}} \approx J_{\text{train}}$
High Variance	Low	High	$J_{\text{cv}} \gg J_{\text{train}}$
Just Right	Low	Low	$J_{\text{cv}} \approx J_{\text{train}}$
Both High	High	Much higher	Large gap, J_{train} also high

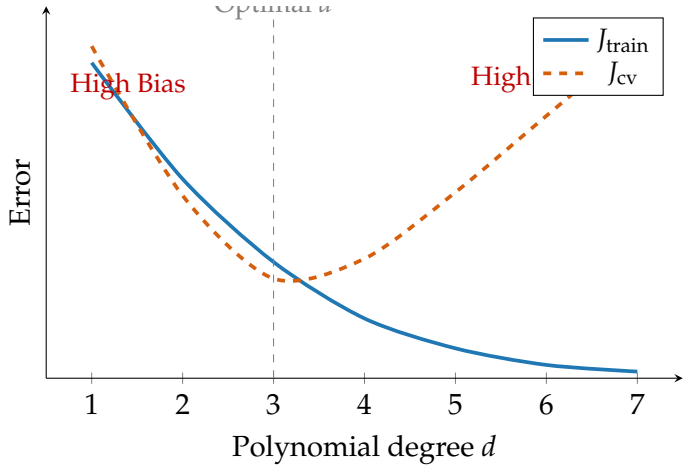


Figure 6.1: J_{train} decreases monotonically with model complexity. J_{cv} follows a U-shaped curve with minimum at the optimal degree.

6.2.2 Regularisation and the Bias–Variance Trade-off

- **Large** λ: forces $\vec{w} \approx \mathbf{0}$, model near constant $\Rightarrow$ **High Bias**.
- **Small** λ: no constraint $\Rightarrow$ **High Variance**.
- **Optimal** λ: balance between fitting data and simplicity.

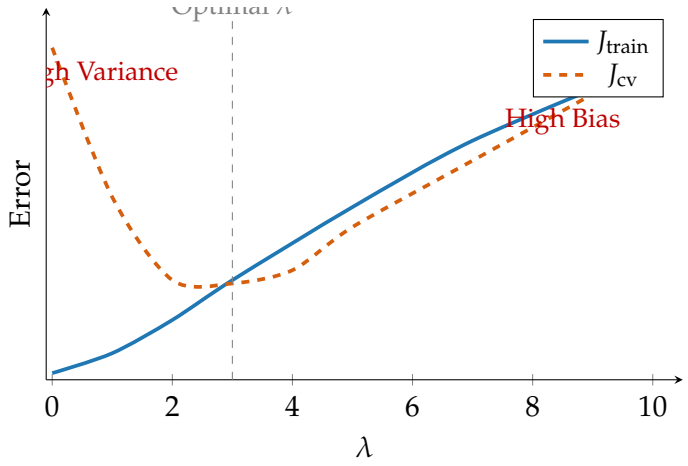


Figure 6.2: As λ increases, J_{train} rises and J_{cv} forms a U-shape. The valley identifies the optimal regularisation strength.

6.2.3 Baseline Performance and the Gap Framework

Comparing J_{train} in isolation can be misleading — many tasks have an irreducible noise floor.

Definition 6.2.3 (Bias gap and variance gap)**.**

$$\text{Bias Gap} = J_{\text{train}} - \text{Baseline}, \qquad \text{Variance Gap} = J_{\text{cv}} - J_{\text{train}}.$$

Large bias gap $\Rightarrow$ High Bias. Large variance gap $\Rightarrow$ High Variance.

Example 6.2.1 (Speech recognition)**.** $J_{\text{train}} = 10.8\%$, $J_{\text{cv}} = 14.8\%$, human baseline $= 10.6\%$.

- Bias Gap = 0.2% (small).
- Variance Gap = 4.0% (large).
- Conclusion: **High Variance** problem.

6.2.4 Debugging Strategies

Diagnosis	Recommended action
High Variance	Get more training examples
	Reduce feature set
	Increase λ
High Bias	Add more / better features
	Add polynomial features
	Decrease λ

6.2.5 Neural Networks and Bias–Variance

A sufficiently large, well-regularised neural network can be made effectively low-bias. The iterative workflow:

1. Check J_{train} vs. baseline. If high $\Rightarrow$ High Bias: **use a bigger network** (more layers/units).
2. Check J_{cv}. If $J_{\text{cv}} \gg J_{\text{train}} \Rightarrow$ High Variance: **get more data**.
3. Repeat until both errors are acceptably low.

```
from tensorflow.keras.regularizers import L2
model = Sequential([
    Dense(units=25, activation='relu', kernel_regularizer=L2(0.01)),
    Dense(units=15, activation='relu', kernel_regularizer=L2(0.01)),
    Dense(units=1,  activation='sigmoid', kernel_regularizer=L2(0.01)
        ),
])
```

Listing 6.1: L_2 regularisation in Keras

6.3 Machine Learning Development Process

6.3.1 The Iterative Development Loop

Building a production ML model is rarely linear. It follows a continuous cycle:

1. **Choose architecture**: model family, features, hyperparameters.
2. **Train model**: implement and train.
3. **Run diagnostics**: bias–variance analysis and error analysis. Use findings to loop back.

6.3.2 Data Augmentation and Synthesis

Definition 6.3.1 (Data augmentation)**. Data augmentation** creates new training examples (x', y) by applying structure-preserving transformations to existing examples (x, y).

- **Computer vision**: rotation, cropping, mirroring, contrast changes, grid-based warping.
- **Speech**: adding background noise, applying filters to simulate poor phone connections.

Key rule: Augmentation distortions must be *representative of real test-set noise*. Purely random pixel noise provides little benefit.

Definition 6.3.2 (Data synthesis)**. Data synthesis** generates entirely new training examples from scratch (e.g. rendering text in diverse fonts for OCR training).

6.3.3 Transfer Learning

Definition 6.3.3 (Transfer learning)**. Transfer learning** reuses a model pre-trained on a large related task as the starting point for a new, data-scarce task.

Standard two-step workflow:

1. **Supervised pre-training**: train a large network on a massive labelled dataset (e.g. 1 million ImageNet images).
2. **Fine-tuning**: replace the output layer for the target task. Then:
    - **Frozen layers (Option 1)**: train only the new output layer. Best for very small new datasets ($\approx$ 50–100 examples).
    - **Full fine-tuning (Option 2)**: retrain all layers using pre-trained weights as initialisation. Best for moderate-to-large new datasets.

Transfer learning works because neural networks learn a hierarchy of features: early layers detect universal low-level patterns (edges, corners) that are task-agnostic.

6.3.4 Ethics and Fairness

As ML systems impact billions of users, **fairness**, **bias detection**, and **ethical development** are mandatory parts of the lifecycle. A framework:

I. **Team diversity**: assemble diverse teams.

II. **Literature review**: consult industry guidelines and standards.

III. **Pre-deployment auditing**: measure performance across demographic subgroups.

IV. **Mitigation and monitoring**: prepare rollback strategy; monitor for real-world harms post-deployment.

6.4 Skewed Datasets

6.4.1 The Problem with Accuracy on Skewed Data

Definition 6.4.1 (Skewed dataset)**.** A dataset is **skewed** when the ratio of positive to negative examples is far from 50/50 (e.g. 99% negative, 1% positive).

A classifier that *always* predicts negative achieves 99% accuracy on such a dataset while providing zero useful predictions. This motivates **Precision** and **Recall**.

6.4.2 Confusion Matrix, Precision, and Recall

		Actual Class	
		Positive ($y = 1$)	Negative ($y = 0$)
Predicted	Positive ($\hat{y} = 1$)	True Positive (TP)	False Positive (FP)
	Negative ($\hat{y} = 0$)	False Negative (FN)	True Negative (TN)

$$\text{Precision} = \frac{TP}{TP + FP}, \qquad \text{Recall} = \frac{TP}{TP + FN}. \tag{6.3}$$

A classifier that always predicts negative has Recall $= 0$, correctly exposing its uselessness.

6.4.3 The Precision–Recall Trade-off

Adjusting the decision threshold τ navigates the trade-off:

- Higher τ (0.7–0.9): higher precision, lower recall. Appropriate when false positives are costly.
- Lower τ (0.3): higher recall, lower precision. Appropriate when false negatives are costly (e.g. missing a deadly disease).

6.4.4 The F1 Score

A single composite score combining precision and recall. The *harmonic mean* penalises extreme imbalance:

$$F_1 = \frac{2PR}{P + R}. \tag{6.4}$$

A high F_1 requires *both* P and R to be high.

Algorithm	P	R	**Arith. mean**	F_1
Algorithm 1	0.50	0.40	0.450	**0.444**
Algorithm 2	0.70	0.10	0.400	0.175
Algorithm 3	0.02	1.00	0.510	0.039

Algorithm 3 achieves the highest arithmetic mean (0.51) but its F1 of 0.039 correctly reveals it as a poor classifier (predicts positive for everything).

Chapter Summary

- **Diagnostics** replace guessing with evidence. The three-way split (train/CV/test) enables unbiased model selection.
- **Bias** (underfitting) and **variance** (overfitting) are the two primary failure modes, diagnosed by comparing J_{train} vs. J_{cv} against a baseline.
- The iterative development loop (architecture $\rightarrow$ train $\rightarrow$ diagnose) is the standard workflow. Large, well-regularised networks tend to be low-bias; more data cures high variance.
- Data augmentation, synthesis, and transfer learning address data scarcity.
- **Skewed datasets** require Precision, Recall, and F_1 score rather than raw accuracy.
- Ethics and fairness must be embedded throughout the development lifecycle.

Chapter 7

Decision Trees

Decision trees are among the most powerful and interpretable machine learning algorithms in widespread use. Unlike linear models, they naturally capture complex non-linear relationships without requiring feature scaling or distributional assumptions, and they serve as the building block for advanced ensemble methods — Random Forests and XGBoost — that consistently rank among the top-performing algorithms on structured tabular data.

Remark 7.0.1. Decision trees are particularly effective for **tabular (structured) data**. For unstructured data (images, audio, raw text), neural networks are generally preferred.

7.1 Decision Tree Basics

7.1.1 Dataset Representation

Consider classifying animals as cats or not, given three observable features:

- x_1: Ear Shape — Pointy or Floppy
- x_2: Face Shape — Round or Not Round
- x_3: Whiskers — Present or Absent
- $y \in \{0, 1\}$: 1 =Cat, 0 =Not Cat

Decision trees can handle both categorical and continuous features. For categorical features with more than two values, one-hot encoding (Section 7.2.5) converts each category into a binary indicator.

7.1.2 Anatomy of a Decision Tree

A decision tree is a hierarchical, acyclic graph encoding a sequence of if–then–else rules.

Root Node The topmost node. Receives the full training dataset.

Internal Nodes Test a specific feature and route examples to child branches based on the outcome.

Leaf Nodes Terminal nodes storing a class label (classification) or numeric value (regression).

The **depth** of a node is the number of edges from the root. The root is at depth 0.

7.1.3 Inference

To classify a new example:

1. Start at the root node.
2. Evaluate the feature tested at the current node.
3. Follow the matching branch.
4. Repeat at each subsequent node.
5. Return the prediction stored in the reached leaf.

Inference is $O(d)$ in tree depth d — very efficient.

7.2 Decision Tree Learning

7.2.1 The Recursive Splitting Algorithm

Tree construction proceeds top-down, greedy, and recursively:

1. Place the entire training set at the root.
2. Select the feature that maximises Information Gain.
3. Split examples into subsets by feature value.
4. Recursively repeat on each child subset.
5. Apply stopping criteria to decide when a node becomes a leaf.

7.2.2 Entropy: Measuring Impurity

At any node, the training subset may mix multiple classes. The degree of mixing is called **impurity**. **Entropy** is the standard measure:

$$H(p_1) = -p_1 \log_2 p_1 - (1 - p_1) \log_2 (1 - p_1), \tag{7.1}$$

where p_1 is the positive-class fraction. Convention: $0 \log_2 0 \equiv 0$.

Properties:

- $H(p_1) = 0$ when $p_1 \in \{0, 1\}$ (pure node).
- $H(p_1) = 1$ when $p_1 = 0.5$ (maximum uncertainty).
- H is concave and symmetric, achieving its maximum at $p_1 = 0.5$.

Composition	p_1	$H(p_1)$	**Impurity**
6 cats, 0 dogs	1.000	0.000	None (pure)
5 cats, 1 dog	0.833	0.650	Low
4 cats, 2 dogs	0.667	0.918	Moderate
3 cats, 3 dogs	0.500	1.000	Maximum
0 cats, 6 dogs	0.000	0.000	None (pure)

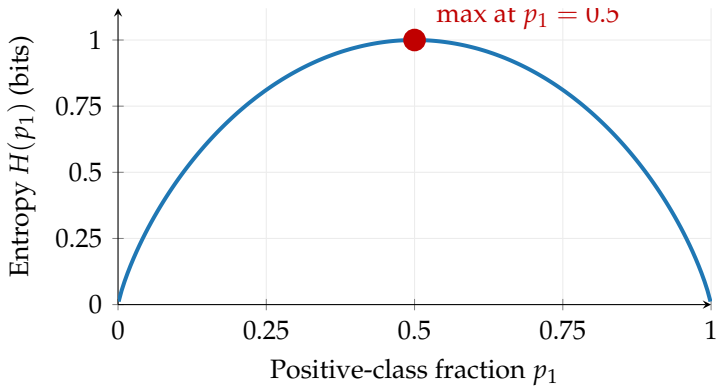


Figure 7.1: Binary entropy $H(p_1)$. It peaks at 1 bit (maximum uncertainty) when the two classes are equally likely, and vanishes at pure nodes.

7.2.3 Information Gain

Let n be the number of examples at the current node, n_L and n_R those reaching the left and right children. The **Information Gain** is:

$$\mathrm{IG}(f) = H(p_1) - \left[w^L H(p_1^L) + w^R H(p_1^R)\right], \quad w^L = \frac{n_L}{n},\ w^R = \frac{n_R}{n}. \tag{7.2}$$

IG is always non-negative (by convexity of entropy) and equals zero when the split provides no useful information.

Example 7.2.1. Root: 10 examples, 5 cats, 5 dogs $\Rightarrow H(p_1) = 1$.

Ear Shape: Left (5 ex., 4 cats): $H = 0.72$. Right (5 ex., 1 cat): $H = 0.72$. IG $= 1 - [0.5(0.72) + 0.5(0.72)] =$ **0.28**.

Face Shape: IG $\approx$ **0.03**.

Whiskers: IG $\approx$ **0.12**.

Decision: select *Ear Shape* (largest IG).

7.2.4 Stopping Criteria

1. **Perfect purity**: stop when $H = 0$.
2. **Maximum depth**: stop at depth $d_{\max}$.
3. **Minimum IG**: stop if best IG $< \epsilon_{\mathrm{IG}}$.
4. **Minimum node size**: stop if $n < n_{\min}$.

7.2.5 One-Hot Encoding

Definition 7.2.1 (One-Hot Encoding)**.** A categorical feature x with k values is replaced by k binary features $b_1, \ldots, b_k$, where $b_j = 1$ if x equals the j-th category.

Example 7.2.2. Ear Shape $\in$ {Pointy, Floppy, Oval}:

Shape	Is Pointy?	Is Floppy?	Is Oval?
Pointy	1	0	0
Floppy	0	1	0
Oval	0	0	1

After OHE, every feature is binary and the standard IG-based splitting applies.

7.2.6 Regression Trees

A **regression tree** predicts a continuous target. A regression leaf stores the **mean** of the target values of all training examples it receives:

$$\hat{y}_{\text{leaf}} = \frac{1}{|\mathcal{S}_{\text{leaf}}|} \sum_{i \in \mathcal{S}_{\text{leaf}}} y_i.$$

The splitting criterion is **variance reduction**:

$$\Delta\sigma^2 = \sigma^2_{\text{root}} - \left[\frac{n_L}{n}\,\sigma^2_L + \frac{n_R}{n}\,\sigma^2_R\right]. \tag{7.3}$$

Aspect	**Classification tree**	**Regression tree**
Target y	Discrete label	Continuous number
Leaf prediction	Majority class	Mean of examples
Split criterion	Information Gain	Variance Reduction

7.3 Tree Ensembles

7.3.1 Motivation: Instability of Single Trees

A single decision tree has **high variance**: a small perturbation in the training data can lead to a completely different tree structure. The solution is to combine many diverse trees.

7.3.2 Bootstrapping and Bagging

Bootstrapping: draw m examples with replacement from a training set of size m. The resulting sample contains some examples multiple times and omits others. On average, each bootstrap sample includes $\approx 63.2\%$ of the unique examples.

Bagging (Bootstrap Aggregating): train an independent decision tree on each of B bootstrap samples, then aggregate predictions by majority vote (classification) or averaging (regression).

Hyperparameter B: typical values $B \in [64, 128]$. Increasing B never hurts accuracy but yields diminishing returns beyond ≈ 128.

7.3.3 Random Forests

In a bagged ensemble all trees share the same full feature set; if one feature is a strong predictor, nearly every tree will use it as the root split, making trees highly correlated.

Random Forest adds a second source of randomness: at each node, only a random subset of $k < n$ features is evaluated:

$$k = \lfloor \sqrt{n} \rfloor \quad \text{(standard for classification).} \tag{7.4}$$

This *decorrelates* the trees, amplifying the variance-reduction benefit of averaging.

7.3.4 Boosting and XGBoost

Bagging trains trees *in parallel* on independent subsamples. **Boosting** trains trees *sequentially*: each new tree focuses on examples the current ensemble handles poorly. Key insight: concentrate effort on weaknesses, not strengths (like deliberate practice).

XGBoost (eXtreme Gradient Boosting) frames boosting as gradient descent in function space. Key engineering features:

- Built-in L_1 and L_2 regularisation on leaf weights.
- Efficient $O(n \log n)$ split-finding via quantile sketches.
- CPU cache-aware computation.
- Native handling of missing values.

```python
from xgboost import XGBClassifier, XGBRegressor

# Classification
clf = XGBClassifier(n_estimators=100, max_depth=6,
                    learning_rate=0.1, random_state=42)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

# Regression
reg = XGBRegressor(n_estimators=100, max_depth=6,
                   learning_rate=0.1, random_state=42)
reg.fit(X_train, y_train)
```

Listing 7.1: XGBoost for classification and regression

7.3.5 Decision Trees vs. Neural Networks

Criterion	Decision Trees/Ensembles	Neural Networks
Best data type	Tabular/structured	Unstructured, mixed
Training speed	Fast	Slow (large models)
Interpretability	High (single trees)	Low (black box)
Transfer learning	Not available	Yes
Feature scaling	Not required	Recommended
Recommended tool	XGBoost, Random Forest	PyTorch, TensorFlow

Practical guidance:

- Use XGBoost/Random Forest for tabular data, fast iteration, or when interpretability is required.
- Use neural networks for images, text, audio, video, or mixed modalities.

Chapter Summary

Decision Tree Basics: hierarchical if–then–else structure traversed in $O(d)$ time; handles categorical and continuous features.

Learning algorithm: recursive top-down splitting that maximises Information Gain $\text{IG}(f) = H(p_1) - [w^L H(p_1^L) + w^R H(p_1^R)]$ at each node. Regression trees replace entropy with variance reduction.

Tree Ensembles: bagging (bootstrap + average) reduces variance. Random Forests add feature randomisation to decorrelate trees. XGBoost uses sequential boosting with gradient descent in function space, achieving state-of-the-art performance on structured data.

Chapter 8

Unsupervised Learning

8.1 Clustering

8.1.1 Overview and Motivation

Clustering is one of the foundational unsupervised learning algorithms. Its goal is to automatically discover patterns, groupings, or structures within a dataset — without any human-provided labels.

Definition 8.1.1 (Cluster)**.** A **cluster** is a subset of training examples such that points within the subset are more similar to one another than to points outside it.

Real-world applications:

1. **News aggregation**: grouping articles by topic.
2. **Market segmentation**: identifying distinct customer groups.
3. **Genomics**: grouping individuals by genetic expression data.
4. **Astronomy**: grouping celestial bodies into coherent structures.
5. **Image compression**: representing an image using K colours.

Feature	Supervised learning	Clustering
Data format	(x, y) pairs with labels	x only
Goal	Learn f s.t. $f(x) \approx y$	Discover groups
Error signal	Labelled ground truth	Cost / distortion function

8.1.2 The K-Means Algorithm

K-means partitions m examples into K clusters by alternating two steps.

Initialisation. Randomly select K distinct training examples and place initial centroids $\mu_1, \ldots, \mu_K$ at those locations.

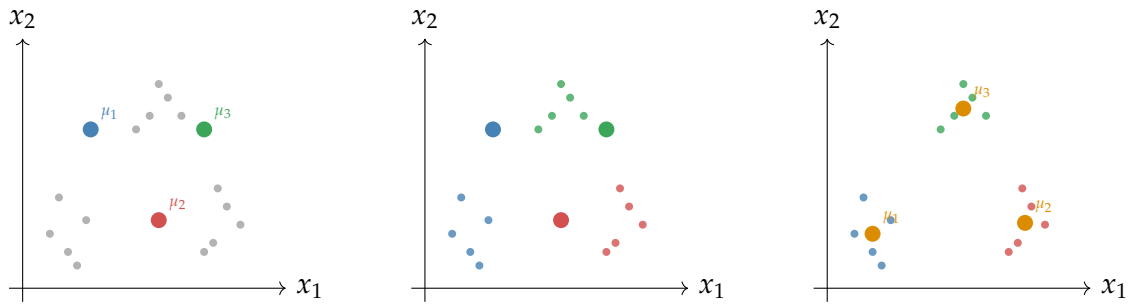
Step 1 — Cluster assignment.

$$c^{(i)} := \arg\min_k \|x^{(i)} - \mu_k\|^2.$$

Step 2 — Centroid update.

$$\mu_k := \frac{1}{|\mathcal{C}_k|} \sum_{i:\, c^{(i)}=k} x^{(i)}.$$

If $\mathcal{C}_k = \emptyset$, eliminate that cluster (reduce K by one).



(a) Initialisation **(b)** After assignment **(c)** Converged

Figure 8.1: K-means with $K = 3$. Coloured points are cluster assignments; orange diamonds are centroids.

8.1.3 The K-Means Cost Function

K-means minimises the **distortion function**:

$$J = \frac{1}{m} \sum_{i=1}^{m} \|x^{(i)} - \mu_{c^{(i)}}\|^2.$$

Each step provably decreases J or leaves it unchanged, so convergence is guaranteed. An increasing J signals a code bug.

Multiple initialisations. Run K-means $R = 50$–1000 times with different random initialisations; keep the solution with the lowest J. This is especially important for small K.

8.1.4 Choosing K

Elbow method. Plot J vs. K; look for a sharp bend. Often unreliable because many real-world curves lack a clear elbow.

Downstream purpose. Choose K based on the end objective (e.g. S/M/L for T-shirt sizing, or quality–compression trade-off for image compression).

8.2 Anomaly Detection

8.2.1 Introduction and Types of Anomalies

Anomaly detection identifies data points that deviate significantly from normal behaviour.

Definition 8.2.1 (Anomaly). x_{test} is anomalous if $p(x_{\text{test}}) < \epsilon$ under a model of normal data.

Three canonical types:

- **Point anomalies**: a single point far from the distribution (e.g. a credit card transaction of \$15,000 for a customer spending \$500/month).
- **Contextual anomalies**: normal in absolute terms but abnormal in context (e.g. 85°F in January in New York).
- **Collective anomalies**: a group of points individually normal but anomalous together (e.g. thousands of packets from one IP in seconds, suggesting a DoS attack).

Applications: fraud detection, cybersecurity, manufacturing quality control, healthcare monitoring, data centre monitoring.

8.2.2 The Gaussian Density Model

The dominant framework: fit a Gaussian to each feature, then flag examples with very low joint probability.

If $X \sim \mathcal{N}(\mu, \sigma^2)$, its PDF is:

$$p(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\,\sigma} \exp\Big(-\frac{(x-\mu)^2}{2\sigma^2}\Big).$$

Maximum likelihood estimates from training data:

$$\hat{\mu} = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}, \qquad \hat{\sigma}^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \hat{\mu})^2.$$

8.2.3 Multi-Dimensional Algorithm

Under the independence assumption, the joint density factors as:

$$p(\vec{x}) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2).$$

Algorithm.

Step 1: Choose n informative features.

Step 2: Fit μ_j and σ_j^2 for each feature j on the training set.

Step 3: For a new example: compute $p(\vec{x})$; flag as anomaly if $p(\vec{x}) < \epsilon$.

8.2.4 Evaluation on Skewed Data

Use precision, recall, and F_1 (not accuracy). Choose ϵ to maximise F_1 on the cross-validation set.

8.2.5 Anomaly Detection vs. Supervised Learning

Feature	Anomaly Detection	Supervised Learning
Positive examples	Very few (0–20)	Many
Anomaly types	Diverse, unpredictable	Recurring, resembles training data
Use case	Fraud, novel defects	Spam, known defects

8.2.6 Feature Engineering Tips

- Apply $\log(x)$ or x^p transforms if a feature is heavily skewed.
- Create ratio features (e.g. CPU load / network traffic) to capture anomalous relationships between features.
- Perform error analysis: inspect missed anomalies to design new features.

Chapter Summary

Clustering. K-means alternates between assignment and centroid update, minimising the distortion $J = \frac{1}{m}\sum \|x^{(i)} - \mu_{c^{(i)}}\|^2$. Run multiple random initialisations. Choose K based on downstream purpose.

Anomaly detection. Fits a Gaussian density model to normal data and flags examples with $p(\vec{x}) < \epsilon$. Evaluation uses F_1; choose ϵ on the CV set. Use anomaly detection when anomaly types are diverse and few positive examples exist; use supervised learning when they are recurring and plentiful.

Chapter 9

Recommender Systems

Recommender systems are among the most commercially successful applications of machine learning. They power suggestions on streaming platforms such as Netflix and Spotify, product recommendations on Amazon, and content feeds of social media platforms. At their core, they learn from past interactions between users and items to predict which items a user is most likely to enjoy next.

Formally: n users, m items, rating matrix $\mathbf{R} \in \mathbb{R}^{n \times m}$ where r_{ui} is user u's rating of item i. Because most users rate only a tiny fraction of items, this matrix is extremely *sparse*. The task is to infer the missing entries.

9.1 Collaborative Filtering

9.1.1 Core Idea

Collaborative filtering's fundamental assumption: *users who agreed in the past tend to agree in the future.* No item content information is required.

9.1.2 Memory-Based CF

User-Based CF

Find users similar to the target user; use their ratings to predict:

$$\hat{r}_{ui} = \bar{r}_u + \frac{\sum_{v \in \mathcal{N}(u)} \operatorname{sim}(u, v)(r_{vi} - \bar{r}_v)}{\sum_{v \in \mathcal{N}(u)} |\operatorname{sim}(u, v)|}. \tag{9.1}$$

Similarity Measures

Cosine similarity.

$$\operatorname{sim}_{\cos}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|_2 \|\mathbf{b}\|_2}.$$

Pearson correlation.

$$\operatorname{sim}_{\mathrm{P}}(u, v) = \frac{\sum_{i \in \mathcal{I}_{uv}} (r_{ui} - \bar{r}_u)(r_{vi} - \bar{r}_v)}{\sqrt{\sum_i (r_{ui} - \bar{r}_u)^2} \sqrt{\sum_i (r_{vi} - \bar{r}_v)^2}}.$$

9.1.3 Model-Based CF: Matrix Factorisation

Approximate the rating matrix as a product of two low-rank factor matrices:

$$\mathbf{R} \approx \mathbf{U}\mathbf{V}^\top, \tag{9.2}$$

$\mathbf{U} \in \mathbb{R}^{n\times k}$ (user factors), $\mathbf{V} \in \mathbb{R}^{m\times k}$ (item factors), $k \ll \min(n, m)$. Predicted rating:

$$\hat{r}_{ui} = \mathbf{u}_u \cdot \mathbf{v}_i.$$

Learning. Minimise regularised MSE over observed ratings:

$$J(\mathbf{U}, \mathbf{V}) = \frac{1}{|\Omega|} \sum_{(u,i)\in\Omega} (r_{ui} - \mathbf{u}_u \cdot \mathbf{v}_i)^2 + \lambda\Big(\sum_u \|\mathbf{u}_u\|^2 + \sum_i \|\mathbf{v}_i\|^2\Big). \tag{9.3}$$

SGD updates for observation (u, i):

$$e_{ui} = r_{ui} - \mathbf{u}_u \cdot \mathbf{v}_i, \tag{9.4}$$

$$\mathbf{u}_u \leftarrow \mathbf{u}_u + \alpha(e_{ui}\mathbf{v}_i - \lambda\mathbf{u}_u), \tag{9.5}$$

$$\mathbf{v}_i \leftarrow \mathbf{v}_i + \alpha(e_{ui}\mathbf{u}_u - \lambda\mathbf{v}_i). \tag{9.6}$$

Bias terms. An augmented model:

$$\hat{r}_{ui} = \mu + b_u + b_i + \mathbf{u}_u \cdot \mathbf{v}_i,$$

where μ is the global mean, b_u the user bias, b_i the item bias.

9.1.4 Limitations of Collaborative Filtering

- **Cold-start problem**: new users or items with no ratings.
- **Popularity bias**: over-recommends popular items.
- **Filter bubbles**: reinforces existing preferences.

9.2 Implementation Details

9.2.1 Cost Function

Let n_u users, n_m movies. Indicator $r^{(i,j)} = 1$ if user j rated movie i.

$$J = \frac{1}{2} \sum_{(i,j):\, r^{(i,j)}=1} \big(\vec{w}^{(j)} \cdot \vec{x}^{(i)} + b^{(j)} - y^{(i,j)}\big)^2 + \frac{\lambda}{2} \sum_j \sum_k (w_k^{(j)})^2 + \frac{\lambda}{2} \sum_i \sum_k (x_k^{(i)})^2. \tag{9.7}$$

Unlike supervised learning, both user parameters $\{(\vec{w}^{(j)}, b^{(j)})\}$ and item features $\{\vec{x}^{(i)}\}$ are simultaneously optimised.

9.2.2 Mean Normalisation

A new user with no ratings yields $\vec{w}^{(j)} = \mathbf{0}$, $\hat{y}^{(i,j)} = 0$ — unhelpful. Mean normalisation:

$$\mu_i = \frac{\sum_{j:\, r^{(i,j)}=1} y^{(i,j)}}{\sum_j r^{(i,j)}}, \qquad y^{(i,j)}_{\text{norm}} = y^{(i,j)} - \mu_i.$$

Prediction: $\hat{y}^{(i,j)} = \vec{w}^{(j)} \cdot \vec{x}^{(i)} + b^{(j)} + \mu_i$. A new user now gets $\hat{y} = \mu_i$ (average rating) — a sensible default.

9.2.3 Finding Related Items

Learned item features $\vec{x}^{(i)}$ can be used to find similar items:

$$\text{sim}(i, i') = \|\vec{x}^{(i)} - \vec{x}^{(i')}\|_2^2.$$

9.3 Content-Based Filtering

9.3.1 Overview

Content-based filtering recommends items by matching item features to a model of the user's preferences. It does not need other users' data.

9.3.2 User Profile and Prediction

Let $\vec{x}^{(i)} \in \mathbb{R}^d$ be the feature vector for item i. A linear user profile $\vec{\theta}^{(j)}$ is estimated from rating history:

$$\hat{y}^{(i,j)} = \vec{\theta}^{(j)} \cdot \vec{x}^{(i)} + b^{(j)}.$$

Advantages.

- Handles cold-start for new *items* (as long as features are available).
- Explainable: "recommended because you liked action movies."
- No popularity bias.

Limitations.

- Requires rich item features.
- Over-specialisation: keeps recommending similar content.
- User cold-start remains.

9.3.3 Hybrid Systems

Production systems typically combine collaborative and content-based filtering. Google's YouTube recommender uses a two-tower neural network: one tower processes user features (including watch history), the other processes video content features; the final score is the dot product of the two embeddings.

9.4 Principal Component Analysis

9.4.1 Motivation

Real-world data often contain far fewer independent sources of variation than the raw number of features suggests. PCA finds the directions of maximum variance and projects data onto a lower-dimensional subspace.

9.4.2 Mathematical Formulation

Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ be mean-centred. The covariance matrix:

$$\mathbf{\Sigma} = \frac{1}{n-1}\mathbf{X}^\top \mathbf{X}.$$

The l-th principal component is the eigenvector of $\mathbf{\Sigma}$ with the l-th largest eigenvalue λ_l. The projection:

$$\mathbf{Z} = \mathbf{X}\mathbf{U}_k,$$

where $\mathbf{U}_k \in \mathbb{R}^{d \times k}$ contains the top-k eigenvectors.

Equivalently via SVD: $\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^\top$; principal components are the columns of $\mathbf{V}$.

9.4.3 Choosing the Number of Components

Fraction of variance explained:

$$\text{EVR}(k) = \frac{\sum_{l=1}^{k} \lambda_l}{\sum_{l=1}^{d} \lambda_l}.$$

Common heuristic: choose k such that $\text{EVR}(k) \geq 0.95$.

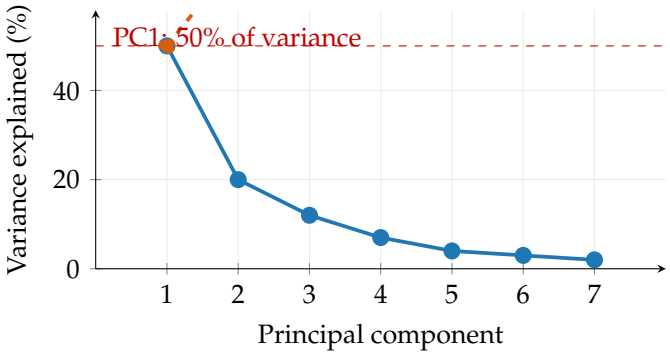


Figure 9.1: Scree plot (blue, left axis) and cumulative variance explained (orange, right axis). Most variance is captured by the first few components.

9.4.4 PCA in Recommender Systems

1. **Dimensionality reduction**: compress high-dimensional content features before feeding into a model.
2. **Latent Semantic Analysis**: truncated SVD directly on the user–item matrix fills in missing ratings: $\hat{\mathbf{R}} = \mathbf{U}_k \mathbf{S}_k \mathbf{V}_k^\top$.
3. **Visualisation**: project learned embeddings to 2D/3D.
4. **Noise reduction**: discard small singular values.

Connection to matrix factorisation. For a fully observed matrix with no regularisation, the rank-k matrix factorisation recovers exactly the truncated SVD:

$$\arg\min_{\mathbf{U},\mathbf{V}} \|\mathbf{R} - \mathbf{U}\mathbf{V}^\top\|_F^2 = \mathbf{U}_k \mathbf{S}_k \mathbf{V}_k^\top .$$

Chapter Summary

1. **Collaborative filtering** exploits shared preferences. Matrix factorisation learns compact latent representations optimised by SGD or ALS.
2. **Mean normalisation** gives sensible default predictions for new users.
3. **Content-based filtering** builds a per-user preference model from item features; handles item cold-start but can over-specialise.
4. **PCA** extracts the principal axes of variation; it underpins matrix factorisation and is used for dimensionality reduction, noise filtering, and latent semantic analysis.
5. Production systems **combine** both paradigms in large-scale deep learning architectures.

Chapter 10

Reinforcement Learning

Reinforcement Learning (RL) is the third major pillar of machine learning, alongside supervised and unsupervised learning. Where supervised learning requires labelled examples of the correct answer and unsupervised learning discovers hidden structure, RL trains an **agent** to take sequential decisions by interacting with an **environment** and receiving feedback as numerical **rewards**. The key feature: the system is told *what to achieve* (via the reward function), not *how to achieve it*.

10.1 Core Concepts

10.1.1 What Is Reinforcement Learning?

Learning proceeds by **trial and error**: the agent tries actions, observes consequences, receives reward/penalty signals, and gradually improves its behaviour to maximise cumulative reward.

Analogy. Training a dog: sitting on command earns a treat (positive reward); misbehaving earns a scolding (negative reward). Over many repetitions the dog learns which behaviours earn treats.

RL vs. supervised learning.

Feature	Supervised Learning	Reinforcement Learning
Input/output	$x \mapsto y$ (label)	$s \mapsto a$ (action)
Data	Expert-labelled dataset	Reward function + experience
Objective	Predict correct label	Maximise cumulative reward
Exploration	None	Essential

A fundamental advantage of RL for control tasks: for complex motor control (robotic locomotion, aerobatic flight) it is often impossible to specify the optimal action in every state. A reward function sidesteps this.

10.1.2 Key Components

1. **State** s: complete description of the current situation.
2. **Action** a: the decision made at each step.
3. **Reward** $R(s)$: scalar feedback signal.

4. **Policy** π: function mapping states to actions, $\pi(s) = a$.
5. **Return** G: discounted cumulative reward (defined below).

10.1.3 The Discount Factor and Return

The **discount factor** $\gamma \in [0, 1)$ encodes the agent's degree of patience.

$$\boxed{G = R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{k+1}.} \tag{10.1}$$

- $\gamma \approx 1$ (patient): future rewards nearly as valuable as immediate ones.
- $\gamma \approx 0$ (myopic): rewards a few steps away are heavily discounted.

Financial analogy. γ mirrors the time value of money: a reward now is more valuable than the same reward after k time steps.

Example (Mars Rover, $\gamma = 0.5$). Starting at state $s = 4$, moving left: reward sequence $[0, 0, 0, 100]$.

$$G = 0 + 0.5(0) + 0.25(0) + 0.125(100) = 12.5.$$

Start	Policy	Reward sequence	Return ($\gamma = 0.5$)
$s = 4$	Always Left	$[0, 0, 0, 100]$	12.5
$s = 2$	Always Left	$[0, 100]$	50.0
$s = 4$	Always Right	$[0, 0, 40]$	10.0
$s = 5$	Always Right	$[0, 40]$	20.0

10.1.4 Markov Decision Processes (MDPs)

The standard mathematical framework for RL.

- **Markov property**: the future depends only on the present state, not on the history.
- **Agent–environment loop**: observe s, select $a = \pi(s)$, environment transitions to s', agent receives $R(s)$. Repeat.

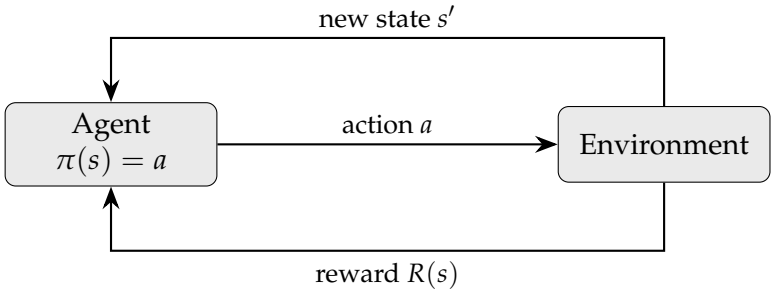


Figure 10.1: The agent–environment interaction loop in a Markov Decision Process.

10.2 State-Action Value Function and the Bellman Equation

10.2.1 The Q-Function

Definition 10.2.1 (Q-function / Action-value function)**.** $Q(s,a)$ is the **total discounted return** obtained by:

1. Starting in state s.
2. Taking action a exactly once.
3. Following the *optimal* policy thereafter.

The optimal policy selects:

$$\pi^*(s) = \arg\max_a Q(s,a). \tag{10.2}$$

State s	Action a	Reward sequence	$Q(s,a)$ ($\gamma = 0.5$)
$s = 2$	Left	$[0, 100]$	$0.5 \times 100 = 50.0$
$s = 2$	Right	$[0, 0, 0, 100]$	$0.5^3 \times 100 = 12.5$
$s = 4$	Left	$[0, 0, 0, 100]$	12.5
$s = 4$	Right	$[0, 0, 40]$	10.0

10.2.2 The Bellman Equation

$$\boxed{Q(s,a) = R(s) + \gamma \max_{a'} Q(s',a').} \tag{10.3}$$

Components.

$R(s)$ Immediate reward for the current step.

γ Discount factor.

$\max_{a'} Q(s',a')$ Best possible future return from the next state.

At terminal states: $Q(s,a) = R(s)$.

Derivation sketch.

$$G = R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots = R(s) + \gamma(R_2 + \gamma R_3 + \cdots) = R(s) + \gamma \max_{a'} Q(s',a'). \quad \square$$

Stochastic environments. When transitions are random, the Bellman equation includes an expectation:

$$Q(s,a) = R(s) + \gamma\, \mathbb{E}[\max_{a'} Q(s',a')]. \tag{10.4}$$

```python
import numpy as np

num_states, gamma = 6, 0.5
rewards   = np.array([100, 0, 0, 0, 0, 40])
q_table   = np.zeros((num_states, 2))   # 2 actions: left=0, right=1
```

```
6
7  def next_state(s, a):
8      return max(0, s-1) if a == 0 else min(5, s+1)
9
10 for _ in range(20):
11     for s in range(1, 5):
12         for a in range(2):
13             s2 = next_state(s, a)
14             terminal = (s2 in [0, 5])
15             r = rewards[s2] if terminal else 0
16             q_table[s, a] = r + (0 if terminal else gamma*q_table[s2
                   ].max())
17
18 policy = {s: ['Left','Right'][q_table[s].argmax()] for s in range
       (1,5)}
19 print(policy)
```

Listing 10.1: Tabular value iteration for the Mars Rover

10.3 Continuous State Spaces and Deep Q-Networks

10.3.1 Continuous States

Real problems require continuous state vectors. A self-driving truck: $s = [x, y, \theta, \dot{x}, \dot{y}, \dot{\theta}]$. A helicopter: 12 state variables. A lookup table is no longer feasible; a neural network must approximate $Q(s, a)$.

10.3.2 Lunar Lander Case Study

State: $s = [x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}, l, r]$ (8D).

Actions: do nothing, fire left thruster, fire main engine, fire right thruster.

Event	Reward	Purpose
Reaching landing pad	$+100$ to $+140$	Primary objective
Moving toward pad	Positive	Progress
Crash	-100	Severe penalty
Soft landing	$+100$	Bonus
Each leg grounded	$+10$	Stability
Main engine firing	-0.3	Fuel cost

Learning objective: maximise $G_t = \sum_{k \geq 0} \gamma^k R_{t+k+1}$, $\gamma = 0.985$.

10.3.3 DQN Architecture

Feed only the state s (8 values) into the network; output *all four Q-values simultaneously*:

- **Input**: 8D state vector.
- **Hidden**: two fully connected layers, 64 ReLU units each.

- **Output**: 4 scalar units, one per action.

Advantage: a single forward pass identifies $\arg\max_a Q(s,a)$, replacing four separate passes.

Training target. Treat as supervised learning: input $= (s,a)$, target $= R(s) + \gamma \max_{a'} Q(s',a')$.

10.3.4 The DQN Algorithm

1. Initialise network Q with random weights.
2. Collect experience: store $(s,a,R(s),s')$ tuples in a **replay buffer**.
3. Sample a mini-batch from the buffer.
4. Compute targets $y_i = R(s_i) + \gamma \max_{a'} Q(s_i',a')$.
5. Train Q_{new} on the mini-batch using MSE loss.
6. Soft-update: $Q \leftarrow Q_{\text{new}}$.
7. Repeat.

10.3.5 The ε-Greedy Policy

With probability ε: take a random action (exploration). With probability $1-\varepsilon$: take $\arg\max_a Q(s,a)$ (exploitation).

Decay schedule: start $\varepsilon = 1.0$, decay toward 0.01 over training.

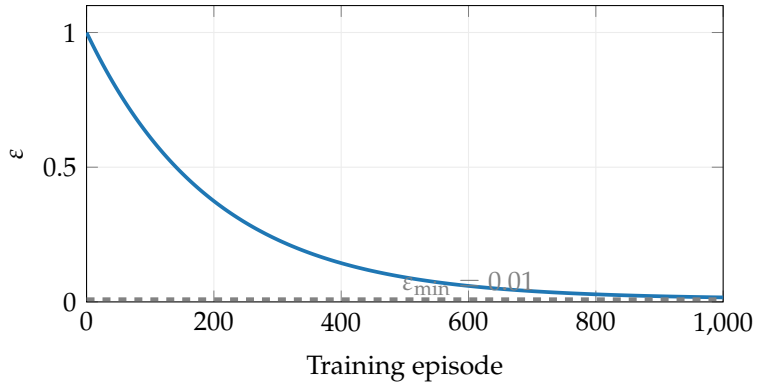


Figure 10.2: Exponential ε-decay from 1.0 toward 0.01 over training.

10.3.6 Mini-Batch and Soft Updates

Mini-batch gradient descent. Sample $m' \ll m$ examples from the replay buffer per update step. Cheaper per step; more steps per wall-clock time; noisy but converges faster in practice.

Soft updates. Blend new parameters gradually:

$$W \leftarrow \tau W_{\text{new}} + (1-\tau)W, \quad b \leftarrow \tau b_{\text{new}} + (1-\tau)b.$$

Typical $\tau = 0.01$: only 1% of new parameters incorporated per step. Prevents a single poor mini-batch from corrupting the policy.

Technique	Purpose	Benefit
Replay buffer	Decouple collection from training	Data diversity
Mini-batching	Updates on random subsets	Training speed
Soft updates	Blend new weights gradually	Stability
ε-greedy	Balance exploration/exploitation	Avoid local optima

10.4 The State of Reinforcement Learning

Current deployment. Despite breakthroughs (AlphaGo, AlphaStar, OpenAI Five), RL deployment in industry is narrower than supervised/unsupervised learning. Most production ML uses labelled data.

The simulation-to-reality gap. RL algorithms are typically developed in simulation. Policies trained there may rely on unrealistic assumptions and fail on real hardware (e.g. sensor noise, actuator delays). Active research in domain randomisation and sim-to-real transfer aims to close this gap.

Future directions.

- **Autonomous control**: the primary framework for sequential decision-making in robotics, logistics, and finance.
- **Alignment research**: RLHF (Reinforcement Learning from Human Feedback) is central to training large language models to follow human intent.

Chapter Summary

1. RL trains an agent to maximise cumulative discounted reward via trial and error, without requiring labelled examples of correct actions.
2. Key components: state s, action a, reward $R(s)$, discount γ, policy π, return $G = \sum \gamma^k R_{k+1}$.
3. An MDP formalises the problem under the Markov property.
4. The Q-function $Q(s,a)$ gives the optimal return for taking action a in state s then acting optimally. $\pi^*(s) = \arg\max_a Q(s,a)$.
5. The **Bellman equation** $Q(s,a) = R(s) + \gamma \max_{a'} Q(s',a')$ is the recursive foundation of all Q-learning algorithms.
6. In stochastic environments: $Q(s,a) = R(s) + \gamma \mathbb{E}[\max_{a'} Q(s',a')]$.
7. Deep Q-Networks use a neural network to represent Q over continuous state spaces.
8. Key engineering: replay buffer, mini-batch updates, soft target updates, ε-greedy exploration with decay.

9. RL deployment remains limited by the simulation-to-reality gap, but RLHF is now critical for aligning large language models.