MACHINE LEARNING

for beginner

(Lecture notes from Machine Learning Specialization Course)

Dat Quoc Truong

Machine Learning for Beginner

Lecture notes from the Machine Learning Specialization Course

Author: Dat Quoc Truong

These notes are compiled from the Machine Learning Specialization course. They serve as a self-contained introductory reference covering both theory and practical implementation.

This document was typeset in LaTeX using Computer Modern fonts, PGFPlots, and Ti*k*Z.

Preface

These lecture notes are designed as a complete, self-contained reference for anyone beginning their journey into machine learning. The notes are organized into ten chapters that progress logically from foundational supervised learning—linear regression and classification—through increasingly sophisticated topics: neural networks and their training, advice for applying machine learning in practice, decision trees, unsupervised learning, recommender systems, and reinforcement learning.

Structure of each chapter. Each chapter opens with a brief motivation and roadmap, builds the mathematical foundations step by step, and illustrates every concept with worked examples, visualizations, and Python code snippets. A summary section at the end of each chapter collects the key takeaways for review.

Prerequisites. High-school algebra and introductory calculus (the concept of a derivative) are sufficient for most material. A little familiarity with linear algebra—vectors, dot products, and matrices—is helpful for Chapters 2 through 5.

Notation. Throughout these notes, vectors are written with an overhead arrow ($\vec{w}$) or in bold ($\mathbf{w}$). Matrices are bold uppercase ($\mathbf{W}$). The i-th training example is $(\mathbf{x}^{(i)}, y^{(i)})$, where the superscript is an example index, *not* an exponent.

Dat Quoc Truong

Contents

CHAPTER 1

Introduction to Machine Learning

Chapter Overview. This chapter introduces the *what* and *why* of machine learning. We survey the three main paradigms—supervised, unsupervised, and reinforcement learning—then develop the simplest supervised model, linear regression, in full detail: the model equation, the squared-error cost function, and the gradient-descent training algorithm.

This chapter introduces the foundations of machine learning. We start 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 — and study its model, its cost function, and its visualization. Finally, we examine *gradient descent*, the optimization 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 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 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 *recognizes and extracts patterns* from data, and then uses those patterns to make predictions, decisions, or descriptions of new inputs.

A classical definition was given by Tom Mitchell:

> *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.*

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 or low-dimensional patterns.
- **Recommender systems.** A specialized family that predicts a user's preferences for items, frequently used by streaming services and online stores.
- **Reinforcement learning.** An agent learns by interacting with an environment, receiving *rewards* or *penalties*, and gradually improving its behavior to maximize cumulative reward.

A simple visualization of how these families relate is shown in Figure 1.1.

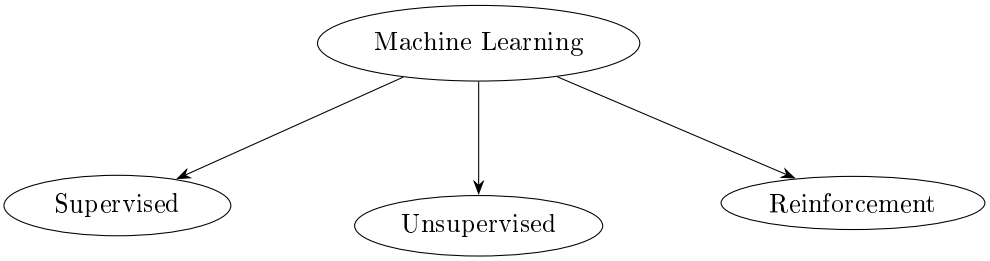


Figure 1.1. Three large families of machine learning algorithms.

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* 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.

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

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

- **Binary** (2 classes): yes/no, 0/1, true/false. Example: *Is this email spam?*

- **Multiclass** (3 or more classes): for instance, classifying a fruit as an *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.
2. Predicting whether a user will buy a product, given features like age, gender, browsing history.
3. Forecasting whether tomorrow will be rainy, using humidity, wind speed, 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, or temperature from time of day.

Figure 1.2 contrasts the two visually.

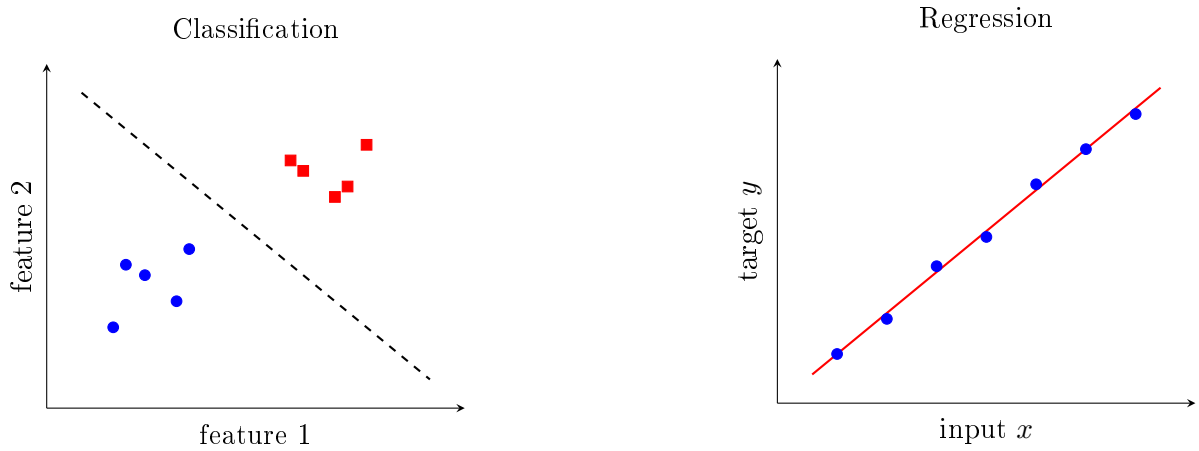


Figure 1.2. Left: classification separates discrete categories. Right: regression fits a continuous function.

The goal of supervised learning is to *automatically* learn the rules that map inputs to outputs by observing many example pairs. 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.

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 on its own.

Common types of unsupervised learning include:

- **Clustering.** Group similar data points together. Examples: grouping news articles in Google News by topic; grouping customers by purchasing behavior; analyzing DNA microarrays.

- **Anomaly detection (outlier detection).** Find unusual data points, e.g. unusual transactions for fraud monitoring.
- **Dimensionality reduction.** Compress the data by representing it with fewer numbers while preserving important structure.

A clustering example is sketched in Figure 1.3.

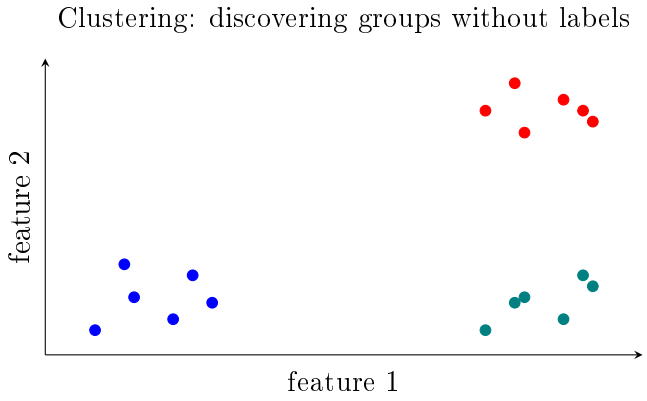


Figure 1.3. A clustering algorithm assigns points to groups (shown by color) without ever being told the correct group.

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 such that $f(x) \approx y$	Discover hidden structure
Typical tasks	Classification, regression	Clustering, anomaly detection, dimensionality reduction
Example	Predicting house price from size	Grouping customers by behavior

1.3 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 inputs.

1.3.1 Setup and Notation

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

size in feet^2 (x, feature)	price in \$1000's (y, target)
2104	460
1416	232
1534	315
852	178
...	...

We use the following standard notation throughout the chapter.

Notation	**Term**	**Meaning**
Training set	training data	The data used to train the model.
x	input variable / feature	E.g. house size.
y	output / target variable	E.g. house price.
m	number of examples	Total number of rows in the dataset.
(x, y)	one training example	A single (input, output) pair.
$(x^{(i)}, y^{(i)})$	i-th training example	Index i is a row index, *not* an exponent.

1.3.2 The Linear Model

Linear regression assumes that 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 called the *parameters* of the model. In machine learning, parameters are precisely the variables we adjust during training in order to make the model fit the data.

Regression vs. classification. A regression model produces a numerical output (infinitely many possible values), whereas a classification model outputs one of a small set of categories (e.g. cat or dog).

1.3.3 The Cost Function

To choose good parameters w and b, we need a way to measure how well the line fits the data. This measure is called a *cost function* (or *loss function*). For linear regression, the most common choice is the **squared-error cost**:

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

The factor $\frac{1}{2m}$ is convenient: $\frac{1}{m}$ averages over examples, and the extra $\frac{1}{2}$ makes derivatives cleaner.

Goal.. Choose w, b to **minimize** $J(w, b)$. When J is small, the model's predictions are close to the actual targets, so the line fits the data well.

1.3.4 Visualizing the Cost Function

To build intuition, temporarily fix $b = 0$ so the model becomes $f_w(x) = wx$. The cost $J(w)$ is then a function of a single variable. As we vary w, the line rotates around the origin, and $J(w)$ traces out a smooth bowl-shaped curve, as shown in Figure 1.4.

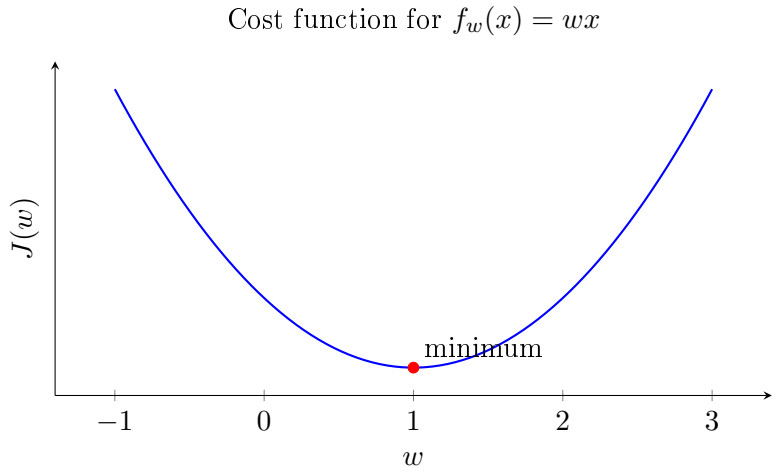


Figure 1.4. For a fixed b, the cost $J(w)$ for linear regression with squared error is a convex (bowl-shaped) function with a unique minimum.

In the general case where both w and b vary, $J(w, b)$ is a function of two variables and looks like a 3D bowl. Slicing the bowl horizontally at constant heights produces a *contour plot*: closed curves where J takes the same value (Figure 1.5).

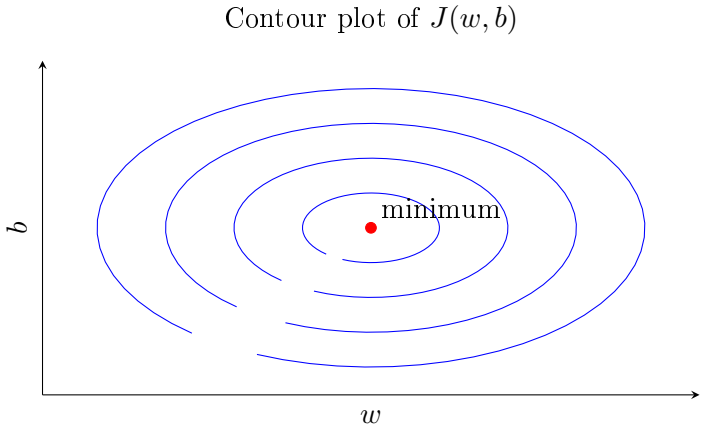


Figure 1.5. Contour plot of a 3D bowl-shaped cost function. Each closed curve is a level set $J(w, b) = c$. The minimum lies at the center.

Worked example.. Suppose we have the tiny training set $\{(1,1),(2,2),(3,3)\}$ and the model $f_w(x) = wx$ (so $b = 0$).

For $w = 1$: predictions equal targets, errors are zero, so $J(1) = 0$.

For $w = 0.5$: predictions are $0.5, 1.0, 1.5$, errors are $-0.5, -1.0, -1.5$. The cost is

$$J(0.5) = \frac{1}{2 \cdot 3}\big(0.25 + 1 + 2.25\big) = \frac{3.5}{6} \approx 0.583.$$

For $w = 0$: predictions are all 0, errors equal the targets, so

$$J(0) = \frac{1}{6}(1 + 4 + 9) = \frac{14}{6} \approx 2.333.$$

So among $w \in \{0, 0.5, 1\}$, the choice $w = 1$ minimizes the cost — which matches the intuition that $y = x$ is the perfect line for this data.

1.4 Training the Model with Gradient Descent

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

1.4.1 The Idea of Gradient Descent

Imagine standing on a hill that represents the cost surface. You want to walk to the lowest point. Without a map, a sensible local strategy is: *look around, find the direction in which the ground drops most steeply, and take a small step in that direction.* Repeat until you can no longer go down.

That is exactly what gradient descent does. It uses the *gradient* (the vector of partial derivatives) of the cost function to identify the direction of steepest *ascent*, and then moves in the *opposite* direction.

Outline..

1. **Initialize** the parameters, often $w = 0$, $b = 0$.
2. **Repeat** (until convergence): update w and b in the direction that reduces $J(w, b)$.
3. **Stop** when the parameters no longer change much — the algorithm has reached a (local) minimum.

1.4.2 The Update Rule

The gradient descent update rule is:

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

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

Here α is the **learning rate**, a small positive number that controls the size of each step.

Important: simultaneous update.. At each iteration, w and b must be updated using the values from the *previous* step. The correct procedure is:

1. Compute the gradients using current w and b.
2. Store the new values in temporaries:

$$\text{tmp}_w = w - \alpha \frac{\partial}{\partial w} J(w,b), \qquad \text{tmp}_b = b - \alpha \frac{\partial}{\partial b} J(w,b).$$

3. Assign $w \leftarrow \text{tmp}_w$, $b \leftarrow \text{tmp}_b$.

1.4.3 Intuition for One Variable

Consider only w, so the update is $w \leftarrow w - \alpha \frac{dJ(w)}{dw}$.

- If the slope at w is **positive**, the cost increases as w increases, so we want to *decrease* w. The rule subtracts a positive number — exactly what we want.
- If the slope is **negative**, the cost decreases as w increases, so we should *increase* w. The rule subtracts a negative number, again moving in the correct direction.

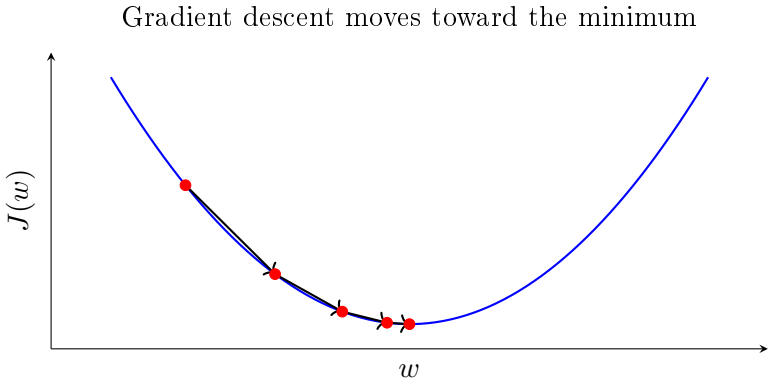


Figure 1.6. Successive gradient descent steps roll down the cost function toward the minimum.

1.4.4 Choosing the Learning Rate

The learning rate α has a strong effect on training.

If α is too small. Each step is tiny, so convergence is very slow and may require an impractical number of iterations.

If α is too large. Steps may *overshoot* the minimum. The cost can oscillate or even grow over time — the algorithm *diverges*.

Figure 1.7 contrasts both behaviors.

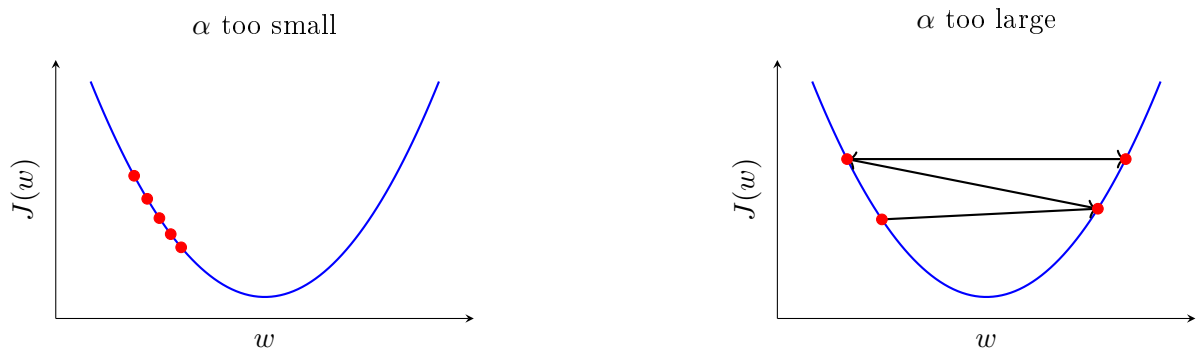


Figure 1.7. Left: a too-small learning rate makes progress painfully slow. Right: a too-large learning rate causes overshoot and possible divergence.

Behavior near a minimum.. At a local minimum, the slope of J is zero, so the update term $\alpha \cdot 0 = 0$, and the parameters stop changing. Interestingly, gradient descent can converge even with a *fixed* learning rate, because as we approach the minimum, the gradient itself shrinks, and so do the steps automatically.

1.4.5 Gradient Descent for Linear Regression

To apply gradient descent to linear regression we need the partial derivatives of the cost function (5.2). Differentiating term-by-term:

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

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

The factor of 2 from the squared-error cancels with the $\frac{1}{2}$ in front, which is why we put it there. The full algorithm is: **Repeat until convergence:**

$$\begin{aligned} w &\leftarrow w - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} \big(f_{w,b}(x^{(i)}) - y^{(i)}\big)\, x^{(i)}, \\ b &\leftarrow b - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} \big(f_{w,b}(x^{(i)}) - y^{(i)}\big). \end{aligned}$$

A useful property: convexity.. For squared-error linear regression, the cost function $J(w,b)$ is *convex* — its 3D shape is a single bowl with no other valleys. As a result, gradient

descent with a sufficiently small learning rate is guaranteed to converge to the *global* minimum, no matter where the parameters are initialized.

1.4.6 "Batch" Gradient Descent

The variant of gradient descent we have described is called **batch gradient descent**, because each step uses *all* m training examples to compute the gradient (all m examples form a single "batch"). Other variants — stochastic gradient descent, mini-batch gradient descent — use one example or a small subset at a time, and they appear later when datasets become too large to use all examples at every step.

1.4.7 Worked Example: Running Gradient Descent

Suppose after training on a housing dataset, the algorithm produces:

$$f(x) = -0.1\,x + 900.$$

This says that, in the units used, every additional square foot decreases the predicted price by \$100, with an intercept of \$900,000 (the values are illustrative). Once trained, we can predict the price for any new house size x by simply plugging it into the formula.

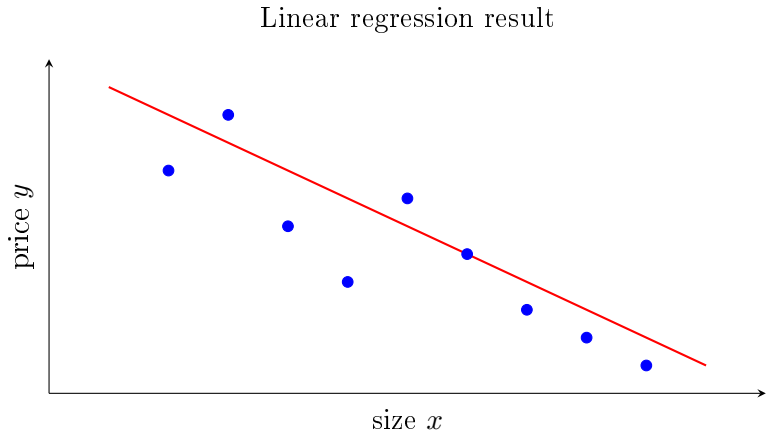


Figure 1.8. The line of best fit produced by gradient descent on a (synthetic) housing dataset. The blue dots are training examples; the red line is the learned model.

Chapter Summary

- Machine learning is the discipline of automatically learning patterns from data.
- The two largest families are *supervised learning* (data has labels, predict y from x) and *unsupervised learning* (no labels, discover structure).
- Supervised problems split into classification (discrete output) and regression (continuous output).

- Linear regression with one variable models the data with a line $f_{w,b}(x) = wx + b$.
- The squared-error cost $J(w, b)$ measures how well the line fits the data; small J means a good fit.
- *Gradient descent* iteratively updates w and b in the direction that decreases J. The learning rate α controls the step size — too small is slow, too large diverges.
- For squared-error linear regression, J is convex, so gradient descent converges to the global minimum.

Exercises

1. For each of the following, decide whether it is a classification or a regression problem.
   (a) Predicting tomorrow's maximum temperature.
   (b) Recognizing whether a handwritten digit is 0–9.
   (c) Estimating the rental price of an apartment from its features.
   (d) Detecting whether a tumor 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?
3. Suppose at some iteration the gradient $\partial J/\partial w$ equals -3 and $\alpha = 0.1$. By how much does w change in the next step? In which direction?
4. Explain in your own words what would happen if the learning rate is set to a very large value such as $\alpha = 100$.
5. True or false: "In supervised learning, the algorithm has to figure out which categories exist by itself." Justify your answer.

CHAPTER 2

Linear Regression with Multiple Variables

Chapter Overview. We extend single-variable linear regression to handle any number of features. Key topics: the vectorised model $\hat{y} = \vec{w} \cdot \vec{x} + b$, computational advantages of NumPy vectorisation, feature engineering and polynomial regression, feature scaling for faster convergence, and diagnosing gradient descent with the learning curve.

2.1 Multiple Linear Regression

In Chapter 1, we studied simple linear regression in which a single feature x was used to predict a scalar target y. Real-world prediction problems almost always involve *more than one* input variable. **Multiple linear regression** generalises the model so that the target y is expressed as a linear combination of n features $x_1, x_2, \ldots, x_n$.

2.1.1 Motivation and Notation

Example 2.1.1 (Housing price prediction)**.** 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.)	300–2,000
x_2	Number of bedrooms	1–5
x_3	Number of floors	1–3
x_4	Age of house (years)	0–80
y	Price ($1,000s)	—

Each row of the dataset is one house; each column (except the last) is a feature.

The following notation is used throughout the chapter.

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 the i-th training example.
- $x_j^{(i)}$ — the value of feature j in the i-th training example.
- An overhead arrow ($\vec{\cdot}$) denotes a vector rather than a scalar.

Remark 2.1.1. In linear algebra, 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, Python's `range(n)` iterates from 0 to $n-1$, not from 1 to n.

2.1.2 The Model

Linear Equation Form

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

$$f_{\vec{w},b}(\vec{x})(\vec{x}) \;=\; w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b. \tag{2.1}$$

Each weight $w_j \in \mathbb{R}$ measures the change in the predicted output for a one-unit increase in x_j, *holding all other features fixed*. The bias b represents the predicted output when every feature is zero; it functions as a *baseline* prediction.

Vectorised Form

Collecting the weights into a row vector $\vec{w} = [w_1, w_2, \ldots, w_n]$ and the features into $\vec{x} = [x_1, x_2, \ldots, x_n]^\top$, Equation (2.1) simplifies to the following dot-product form:

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

The dot product multiplies corresponding pairs of entries and sums the results: $\vec{w} \cdot \vec{x} = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$.

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

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

where the features are those in Example 2.1.1 and prices are 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 cost).

- $w_4 = -0.5$: each additional year of age reduces the price by \$500.
- $b = 80$: the model's baseline is \$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 that can perform many floating-point operations *simultaneously* (in parallel). **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.

Comparing Three Coding Approaches

The following table shows three ways to compute $f_{\vec{w},b}(\vec{x})(\vec{x}) = \vec{w} \cdot \vec{x} + b$:

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 — parallel hardware

Behind the Scenes: Parallelism

Figure 2.1 illustrates the key difference. In the sequential approach, each multiplication $w_j x_j$ is performed as a separate step. In the vectorised approach, a CPU or GPU multiplies *all* pairs simultaneously in a single hardware instruction, then uses a fast adder tree to accumulate the products.

NumPy Implementation

```python
import numpy as np
# 4-feature housing example
w = np.array([0.1, 4.0, -2.0, -0.5])    # weight vector
b = 80.0                                   # bias
x = np.array([1500, 3, 2, 10])             # one training example
# Vectorised prediction (preferred)
f = np.dot(w, x) + b
print(f"Predicted price: ${f:.1f}k")
```

Listing 2.1. Computing a prediction with NumPy

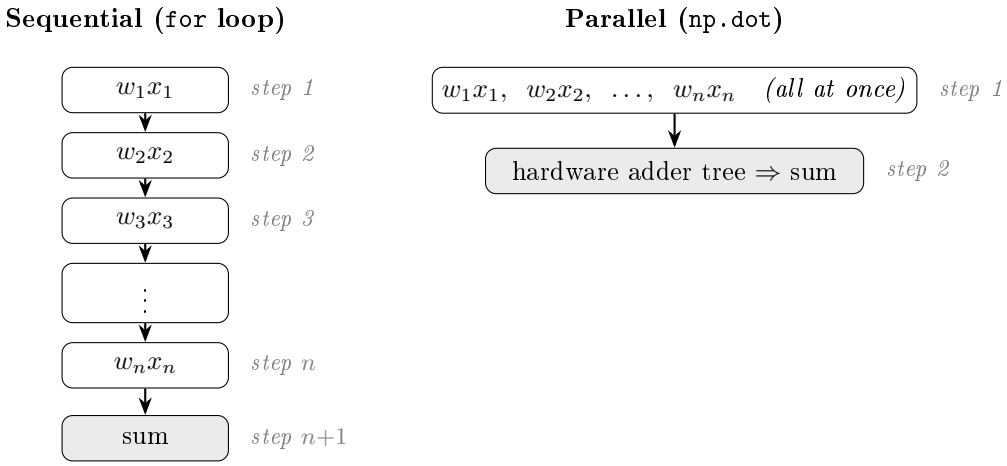


Figure 2.1. Sequential versus parallel computation of $\vec{w} \cdot \vec{x}$. Vectorisation collapses n sequential multiplications into two hardware steps.

Vectorised Gradient Descent Updates

Vectorisation applies equally to the parameter update step. Rather than looping over each weight, all updates are performed in a single array operation:

```
# d : NumPy array of partial derivatives, shape (n,)
# db : scalar partial derivative for bias
alpha = 0.01    # learning rate
w = w - alpha * d     # 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 insight 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.
- x_2: lot depth in metres.

We engineer a new feature:

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

If area is more predictive of price than either dimension individually, the model will assign a large weight w_3, improving accuracy. 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 nonlinear transformations of the original features, we fit *curved* relationships while still using the linear regression framework — because the model remains linear in the *parameters*.

Common Polynomial Model Types

Let x denote house size in square feet. 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.

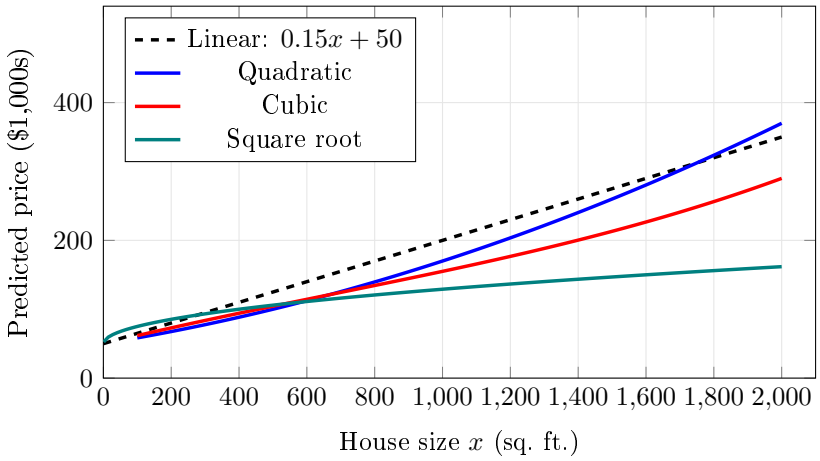


Figure 2.2. Four regression models on hypothetical housing data. The square-root model (teal) rises steeply at first and flattens; the cubic (red) provides a smooth upward trend; the quadratic (blue) eventually turns downward, which may be unrealistic for housing prices.

Feature Scaling is Critical for Polynomial Features

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

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

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

Model Selection

Choosing between competing polynomial models (or deciding which engineered features to include) is a formal process of comparing test-set or cross-validation performance. Scikit-learn provides convenient utilities:

```
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline

pipe = Pipeline([
('poly',   PolynomialFeatures(degree=3, include_bias=False)),
('scaler', StandardScaler()),
('model',  LinearRegression()),
])
pipe.fit(X_train, y_train)
y_pred = pipe.predict(X_test)
```

Listing 2.3. Polynomial regression with Scikit-learn

2.2 Gradient Descent in Practice

2.2.1 Gradient Descent for Multiple Features

The Cost Function

With m training examples and n features, the cost function is the mean squared error:

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

The factor $\frac{1}{2}$ is included for algebraic convenience: it cancels the coefficient of 2 that arises when differentiating the squared term.

Simultaneous Parameter Update Rules

Gradient descent repeatedly updates all parameters *simultaneously*:

$$w_j \;\leftarrow\; w_j - \alpha \, \frac{\partial J(\vec{w}, b)}{\partial w_j}, \qquad j = 1, \ldots, n, \tag{2.7}$$

$$b \;\leftarrow\; b - \alpha \, \frac{\partial J(\vec{w}, b)}{\partial b}, \tag{2.8}$$

where $\alpha > 0$ is the **learning rate**. The partial derivatives are:

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

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

Note that Equation (2.9) differs from its single-feature analogue only by the factor $x_j^{(i)}$. The update for b in Equation (2.10) is the same in both the single- and multiple-feature settings.

Remark 2.2.1 (Simultaneous update)**.** All new parameter values must be computed using the *current* weights before any update is applied. Updating w_1 first and then using the new w_1 to compute the gradient for w_2 is incorrect. In practice, the vectorised implementation naturally enforces simultaneity.

Vectorised Implementation

```python
import numpy as np
def compute_cost(X, y, w, b):
m = X.shape[0]
y_hat = X @ w + b
return (0.5 / m) * np.sum((y_hat - y) ** 2)
def compute_gradient(X, y, w, b):
"""
X   : (m, n) feature matrix
y   : (m,)   target vector
w   : (n,)   weight vector
b   : float  bias
Returns dw (n,), db (float)
"""
m      = X.shape[0]
y_hat  = X @ w + b           # (m,) predictions
err    = y_hat - y           # (m,) residuals
dw     = (X.T @ err) / m     # (n,) weight gradients
db     = err.mean()          # scalar bias gradient
return dw, db
def gradient_descent(X, y, w_init, b_init, alpha, num_iter):
w, b = w_init.copy(), float(b_init)
history = []
for _ in range(num_iter):
dw, db = compute_gradient(X, y, w, b)
w = w - alpha * dw
b = b - alpha * db
history.append(compute_cost(X, y, w, b))
return w, b, history
```

Listing 2.4. 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 chosen carefully	Not required
Computational complexity	$O(k \cdot m \cdot n)$	$O(n^3)$ for matrix inversion
Scalability	Excellent ($n > 10^6$ viable)	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. Some libraries use the Normal Equation internally for small datasets, but expose a gradient-descent-style API to the user.

2.2.2 Feature Scaling

Motivation

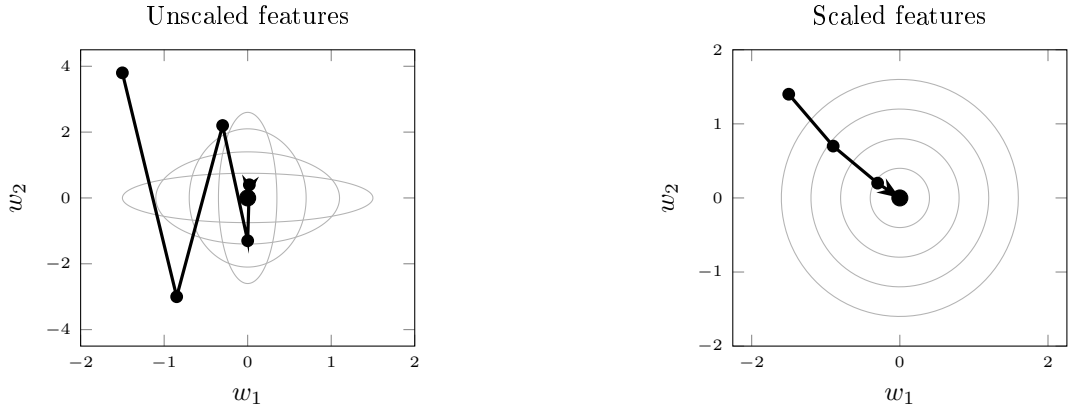
When features have very different numerical ranges, the cost function takes on an elongated, elliptical shape in parameter space. Gradient descent then oscillates along the steep walls of these ellipses rather than heading directly toward the minimum, leading to slow convergence. Scaling features to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

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 in sq. ft. $\approx 1{,}000$) requires only a *small* weight (e.g. $w_1 \approx 0.1$) to produce a reasonable output.
- A feature with a *small* range (e.g. number of bedrooms ≈ 1–5) requires a *large* weight (e.g. $w_2 \approx 50$).

Unequal weight magnitudes lead to cost-function contours with very different curvature along each axis, which is exactly the elliptical geometry that causes slow convergence.



(a) Slow, oscillating path **(b)** Direct, fast convergence

Figure 2.3. Contour plots of $J(w_1, w_2)$. Without scaling (left), features with large ranges produce elongated ellipses and a bouncing gradient path. With scaling (right), the contours are circular and gradient descent converges directly. The dot marks the global minimum.

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 mean of feature j over the training set. Resulting values 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 of feature j. 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
$[-0.3, +0.3]$ or narrower	Yes	Too small; gradients negligible
$[-100, +100]$ or wider	Yes	Dominates gradient
$[0, 0.0001]$	Yes	Extremely compressed scale

Remark 2.2.2. Features do not need to be *perfectly* comparable; they merely need to be *roughly* in the same ballpark. When in doubt, always scale — it virtually never hurts performance and almost always accelerates convergence.

NumPy and Scikit-learn Implementation

```
import numpy as np
from sklearn.preprocessing import StandardScaler
# --- Manual (NumPy) ----
mu    = X_train.mean(axis=0)     # compute on TRAINING set
sigma = X_train.std(axis=0)
X_train_sc = (X_train - mu) / sigma
X_test_sc  = (X_test  - mu) / sigma    # apply SAME statistics to test
# --- Scikit-learn ----
scaler = StandardScaler()
X_train_sc = scaler.fit_transform(X_train)    # fit + transform
X_test_sc  = scaler.transform(X_test)         # transform only (no fit!)
```

Listing 2.5. Z-score normalisation in NumPy and Scikit-learn

Remark 2.2.3. Always compute the mean μ_j and standard deviation σ_j from the *training* set only, then apply those same constants to the test (or validation) set. Fitting a 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 cost $J(\vec{w}, b)$ against the number of gradient descent iterations. It is the primary diagnostic tool for verifying that training is progressing correctly. Key observations

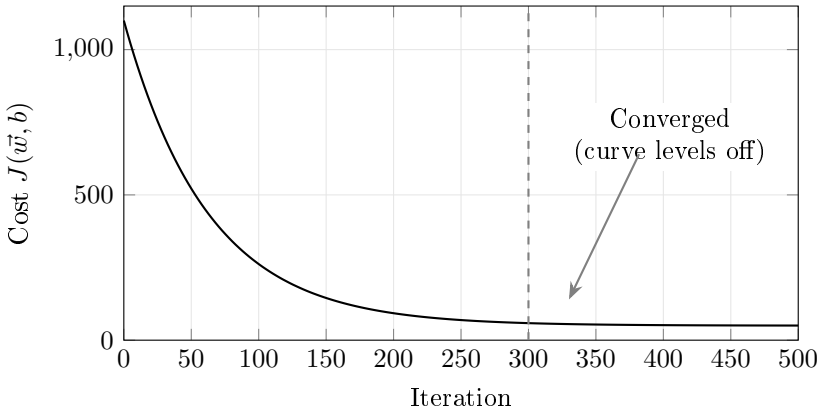


Figure 2.4. A typical learning curve. The cost decreases rapidly at first and levels off as the algorithm converges. If the curve increases or oscillates, the learning rate is too large or there is a bug in the code.

about learning curves:

- A correct implementation 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 signals a learning-rate or code problem.

Automatic Convergence Test

Define a small threshold $\varepsilon > 0$ (e.g. $\varepsilon = 10^{-3}$) and declare convergence when the decrease in cost over one iteration is smaller than ε:

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

Choosing a reliable ε is difficult in practice; visual inspection of the learning curve is often more reliable for identifying subtle bugs or poor hyperparameter choices.

2.2.4 Choosing the Learning Rate

The learning rate α is one of the most consequential hyperparameters in gradient descent.

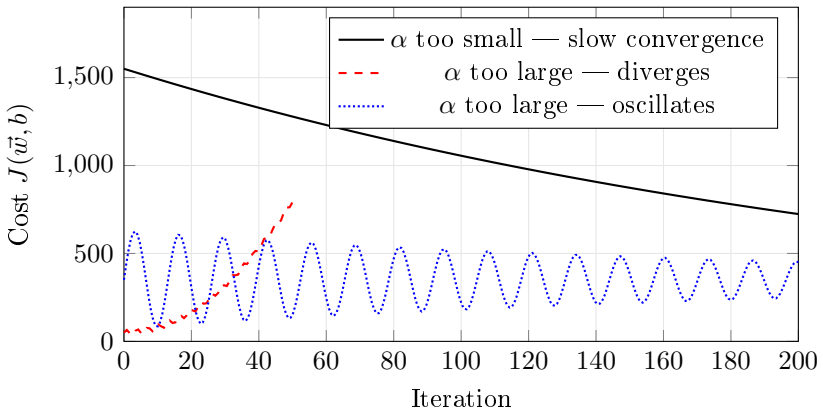


Figure 2.5. Three learning-rate behaviours. A small α converges safely but slowly. A large α causes the cost to either diverge (red) or oscillate (blue) without reaching the minimum.

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. The parameter bounces 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 $+\alpha$ instead of $-\alpha$ in the update rule, which moves parameters *away* from the minimum.

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 in 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.

Practical Checklist Before Running Gradient Descent

1. Scale all features to comparable ranges (Section 2.2.2).

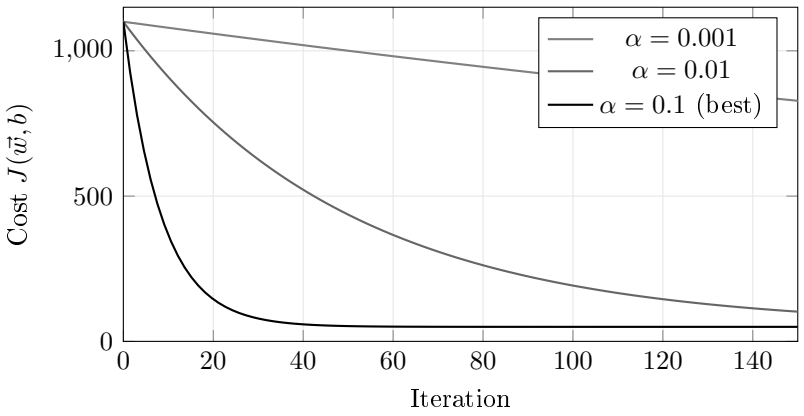


Figure 2.6. Learning curves for three candidate learning rates. The largest stable value ($\alpha = 0.1$) converges the fastest, reaching a near-optimal cost within roughly 50 iterations.

2. Verify that the update rule *subtracts* the gradient: $w_j \leftarrow w_j - \alpha\, \partial J / \partial w_j$.
3. Ensure *simultaneous* updates: compute all gradients using the current parameters before modifying any of them.
4. Plot the learning curve after training and confirm monotonic decrease.

2.2.5 End-to-End Example

The following listing illustrates the complete pipeline — data scaling, gradient descent, and result extraction — on a small toy dataset.

```
import numpy as np
import matplotlib.pyplot as plt

# --- Dataset: 6 houses, 2 features ----

X = np.array([[1200, 3],
[1500, 4],
[900,  2],
[1800, 4],
[1100, 3],
[2000, 5]])     # (m=6, n=2)
y = np.array([250, 320, 190, 380, 230, 420])  # prices in $1000s

# --- Feature scaling (z-score) ----

mu    = X.mean(axis=0)
sigma = X.std(axis=0)
Xs    = (X - mu) / sigma

# --- Initialise ----

w = np.zeros(Xs.shape[1])
b = 0.0
alpha   = 0.1
```

```
n_iter  = 500
history = []

# --- Gradient descent ----

for _ in range(n_iter):
y_hat = Xs @ w + b
err   = y_hat - y
dw    = (Xs.T @ err) / len(y)
db    = err.mean()
w     = w - alpha * dw
b     = b - alpha * db
history.append(0.5 * np.mean(err ** 2))

# --- Results ----

print(f"Weights: {w}")
print(f"Bias:    {b:.2f}")
print(f"Final cost: {history[-1]:.4f}")

# --- Learning curve ----

plt.plot(history)
plt.xlabel("Iteration")
plt.ylabel("Cost J")
plt.title("Learning Curve")
plt.grid(True)
plt.show()
```

Listing 2.6. Complete multiple linear regression pipeline

Chapter Summary

- **Multiple linear regression** models the target as $f_{\vec{w},b}(\vec{x})(\vec{x}) = \vec{w} \cdot \vec{x} + b$, a linear combination of n features plus a bias.
- Each weight w_j captures the marginal effect of feature x_j on the output, holding all other features fixed. A positive w_j indicates a direct relationship; a negative w_j indicates an inverse relationship.
- **Vectorisation** (via `np.dot` and matrix operations) replaces slow sequential loops with parallel hardware instructions, yielding dramatic speedups for large feature counts.
- **Feature engineering** — transforming or combining raw features — can reveal more predictive representations. **Polynomial regression** uses powers and nonlinear functions of features within the linear regression framework to fit curved relationships.
- **Feature scaling** (especially z-score normalisation) maps features to comparable numerical ranges, converting elongated cost-function ellipses into nearly circular contours and greatly accelerating gradient descent.

- The **learning curve** (cost vs. iterations) is the key diagnostic: a correct implementation yields a monotonically decreasing curve that eventually flattens at convergence.
- The **learning rate** α controls the step size. It should be chosen as the largest value that produces stable, monotonic decrease in the cost. Test candidates on a logarithmic grid $(0.001, 0.003, 0.01, \ldots)$.
- The **Normal Equation** provides a closed-form alternative to gradient descent, but is computationally impractical for $n \gtrsim 10{,}000$ and does not generalise beyond linear regression.

CHAPTER 3

Classification

Chapter Overview. Classification assigns inputs to discrete categories. This chapter introduces *logistic regression*—the canonical binary classifier—derives the cross-entropy loss from maximum-likelihood principles, develops the gradient-descent training algorithm, and carefully examines the problem of overfitting and regularisation as its cure.

In the previous chapter we studied *linear regression*, whose goal was to predict a continuous numerical output — for instance, the price of a house or tomorrow's 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* or *classes*. This task is called **classification**.

Classification problems appear everywhere in applied machine learning.

- **Spam detection:** is an incoming email spam or not?
- **Fraud detection:** is a credit-card transaction fraudulent or legitimate?
- **Medical diagnosis:** is a tumour malignant or benign?
- **Image recognition:** which digit (0–9) appears in this image?

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

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

where $y = 1$ denotes the *positive class* (presence of the feature being detected) and $y = 0$ denotes the *negative class* (absence). This naming carries no moral connotation; it merely reflects the presence or absence of the condition of interest.

This chapter is organised as follows. Section 3.1 introduces *logistic regression*, the canonical algorithm for binary classification. Section 3.2 derives the appropriate cost function and motivates the choice of *cross-entropy loss*. Section 3.3 describes gradient descent for optimising the logistic model. Section 3.4 discusses the problem of overfitting and the principal remedies, with special attention to *regularisation*.

3.1 Classification with Logistic Regression

3.1.1 Why Linear Regression Is Inadequate

A natural first instinct is to repurpose the linear regression model $f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$ for classification by introducing a decision threshold: predict $\hat{y} = 1$ whenever $f > 0.5$ and $\hat{y} = 0$ otherwise. While this can occasionally work in simple situations, it suffers from two fundamental problems.

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 trained on this data may yield a reasonable decision 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^* to the right. Points that were previously classified correctly (tumours with size near the old boundary) are now misclassified. Because linear regression is globally sensitive to every training point, one aberrant observation can corrupt the boundary for the rest of the data.

Unbounded output range.. Linear regression produces outputs on $(-\infty, +\infty)$. For a yes/no problem, predicting values such as 1.8 or -0.4 is awkward: these values do not represent probabilities, and interpreting them as such is mathematically unsound. We need a model whose output is *guaranteed* to lie in $[0, 1]$.

3.1.2 The Sigmoid (Logistic) Function

The resolution to both difficulties is to *squash* the linear output through a function that maps $\mathbb{R}$ onto the open interval $(0, 1)$. The standard choice is the **sigmoid function** (also called the *logistic function*), denoted $g(z)$:

Definition 3.1.1 (Sigmoid Function)**.**

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

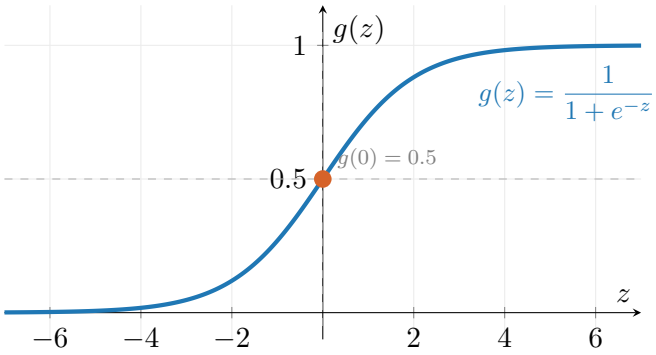


Figure 3.1. The sigmoid (logistic) function $g(z)$. It maps any real number to the interval $(0, 1)$, making it ideal for outputting probabilities. For large positive z, $g(z) \approx 1$; for large negative z, $g(z) \approx 0$.

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 $z = 0$: $g(0) = \frac{1}{1+1} = 0.5$.
- g is smooth, strictly increasing, and S-shaped (sigmoidal).
- $g'(z) = g(z)\big(1 - g(z)\big)$ — a convenient identity for calculus.

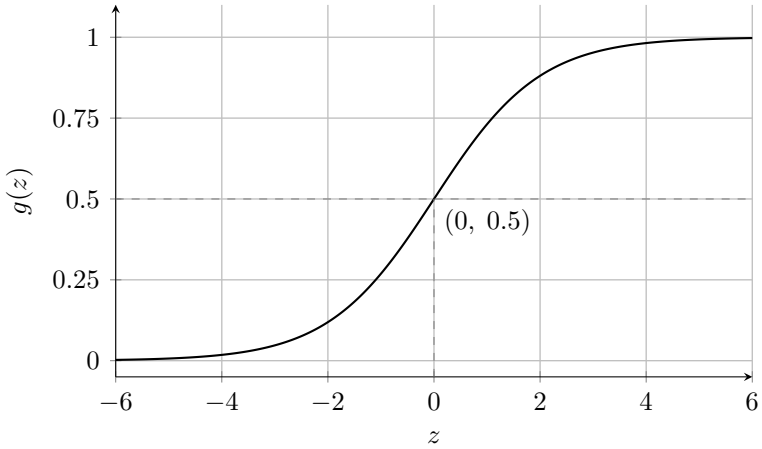


Figure 3.2. The sigmoid function $g(z) = 1/(1+e^{-z})$. It is bounded between 0 and 1 and equals 0.5 at $z = 0$.

3.1.3 The Logistic Regression Model

Logistic regression is built in two stages.

Stage 1 — Linear combination. As in linear regression, compute a weighted sum of the input features:

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

Stage 2 — Sigmoid activation. Pass z through the sigmoid function to obtain the model

output:

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

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 $(\vec{w}, b)$. Since y must be 0 or 1, the complementary probability is

$$P(y = 0 \mid \vec{x};\ \vec{w}, b) = 1 - f_{\vec{w},b}(\vec{x}).$$

Example 3.1.1. Suppose the model is used for tumour classification with a single feature x (tumour size in cm). If $w = 2$ and $b = -5$, then for $x = 3$:

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

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 set to 0.5:

$$\hat{y} = \begin{cases} 1 & \text{if } f_{\vec{w},b}(\vec{x}) \geq 0.5, \\ 0 & \text{otherwise.} \end{cases}$$

Because $g(z) \geq 0.5$ if and only if $z \geq 0$, 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 all input points $\vec{x}$ for which the model is exactly neutral, i.e. $\vec{w} \cdot \vec{x} + b = 0$.

Linear decision boundaries.. With two features x_1 and 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. Let $w_1 = 1$, $w_2 = 1$, $b = -3$. The boundary is $x_1 + x_2 = 3$, a diagonal line in the (x_1, x_2) plane. Points satisfying $x_1 + x_2 > 3$ are classified as $y = 1$; those satisfying $x_1 + x_2 < 3$ as $y = 0$.

Non-linear decision boundaries.. By augmenting the feature vector with polynomial terms (e.g. x_1^2, x_2^2, x_1x_2, ...), the decision boundary in the *original* feature space can be non-linear. For instance, with features (x_1^2, x_2^2) and parameters $w_1 = 1$, $w_2 = 1$, $b = -1$, the boundary becomes:

$$x_1^2 + x_2^2 = 1,$$

a circle of unit radius. Points *outside* the circle are predicted as $y = 1$; points *inside* as $y = 0$. Higher-order polynomials can produce ellipses, figure-eights, and other complex shapes, giving logistic regression great flexibility without abandoning its probabilistic interpretation.

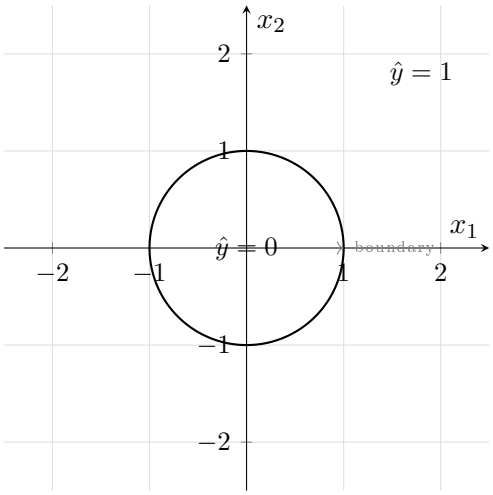


Figure 3.3. Circular decision boundary arising from the polynomial feature map (x_1^2, x_2^2) with $w_1 = w_2 = 1$ and $b = -1$.

3.2 Cost Function for Logistic Regression

3.2.1 Why Squared Error Fails

A natural starting point would be to reuse the mean squared error (MSE) cost from linear regression:

$$J_{\text{MSE}}(\vec{w}, b) = \frac{1}{2m} \sum_{i=1}^{m} \left(f_{\vec{w},b}(\vec{x})^{(i)} - y^{(i)} \right)^2,$$

where $f_{\vec{w},b}(\vec{x})^{(i)} = g(\vec{w} \cdot \vec{x}^{(i)} + b)$ is the sigmoid output.

The difficulty is that composing the non-linear sigmoid function with the squared-error formula produces a *non-convex* cost surface — one with multiple local minima and flat plateaus. Gradient descent may converge to a sub-optimal local minimum rather than the global one, yielding a poor classifier. We therefore need a different cost function that preserves convexity.

3.2.2 Loss vs. Cost: A Key Distinction

Before writing the new cost, we introduce a distinction that will be used throughout the rest of this series.

Definition 3.2.1 (Loss and Cost)**.** The **loss** L measures the penalty on a *single* training example:

$$L\big(f, y\big) = \text{penalty for predicting } f \text{ when the true label is } y.$$

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 logistic loss function piecewise, depending on the true label y:

$$\text{If } y = 1: \quad L = -\log\big(f_{\vec{w},b}(\vec{x})\big), \tag{3.1}$$

$$\text{If } y = 0: \quad L = -\log\big(1 - f_{\vec{w},b}(\vec{x})\big). \tag{3.2}$$

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

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

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

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

The logarithmic shape ensures that the model is *severely penalised* for confident incorrect predictions, while small errors incur only a small penalty. This design choice encodes the desirable asymmetric response to overconfident mistakes.

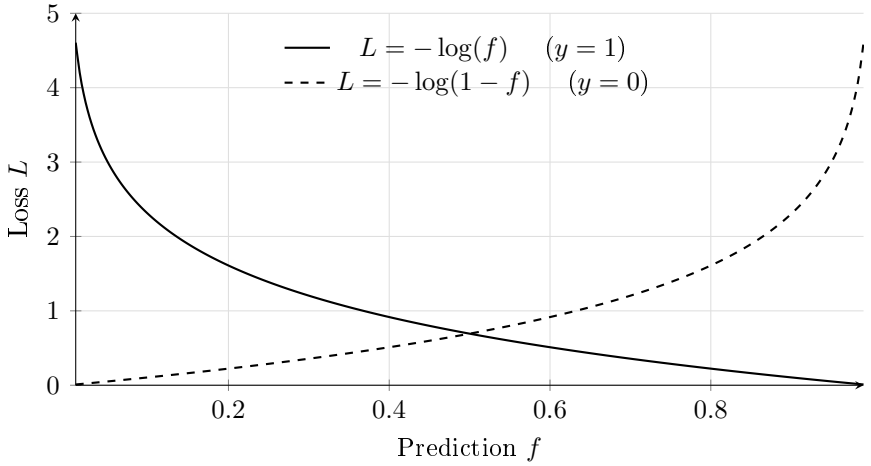


Figure 3.4. Logistic loss functions for $y = 1$ (solid) and $y = 0$ (dashed). Both curves approach $+\infty$ when the prediction is maximally wrong.

3.2.4 The Unified Loss Formula

The piecewise definition can be elegantly collapsed into a single expression by exploiting the binary nature of y:

$$L\big(f_{\vec{w},b}(\vec{x}^{(i)}),\, y^{(i)}\big) = -y^{(i)} \log\big(f_{\vec{w},b}(\vec{x}^{(i)})\big) - \big(1 - y^{(i)}\big) \log\big(1 - f_{\vec{w},b}(\vec{x}^{(i)})\big). \tag{3.3}$$

Proof of equivalence..

- **Case** $y^{(i)} = 1$**:** The factor $(1 - y^{(i)}) = 0$ annihilates the second term, leaving $L = -\log(f_{\vec{w},b}(\vec{x}^{(i)}))$. ✓
- **Case** $y^{(i)} = 0$**:** The factor $y^{(i)} = 0$ annihilates the first term, leaving $L = -\log(1-f_{\vec{w},b}(\vec{x}^{(i)}))$. ✓

3.2.5 The Logistic Regression Cost Function

Averaging the unified loss over all m training examples gives the standard cost function used to train logistic regression:

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

This is commonly called the **cross-entropy loss** or **log-loss**.

Why this specific function?. There are two complementary justifications.

1. **Statistical foundation.** Equation (3.4) is derived from the principle of *Maximum Likelihood Estimation* (MLE). If we model each label $y^{(i)}$ as a Bernoulli random variable with success probability $f_{\vec{w},b}(\vec{x})^{(i)}$, the likelihood of observing the entire training set is

   $$\mathcal{L}(\vec{w}, b) = \prod_{i=1}^{m} \big(f_{\vec{w},b}(\vec{x}^{(i)})\big)^{y^{(i)}} \big(1 - f_{\vec{w},b}(\vec{x}^{(i)})\big)^{1-y^{(i)}}.$$

   Taking the negative log of $\mathcal{L}$ and normalising by m yields exactly $J(\vec{w}, b)$ as in Equation (3.4). Thus, minimising the cross-entropy is equivalent to finding the parameters that make the observed data most probable — a principled and well-understood statistical objective.
2. **Convexity.** Because the sigmoid function is log-concave, the cost $J($ $vw, b)$ defined above is a convex function of ($vw, b)$. This guarantees that 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 wish to find the parameters $(\vec{w}^*, b^*)$ that minimise the cost:

$$(\vec{w}^*, b^*) = \arg\min_{\vec{w}, b} \; J(\vec{w}, b).$$

Once obtained, the trained model can estimate the probability $P(y = 1 \mid \vec{x};\ \vec{w}^*, b^*)$ for any new input $\vec{x}$, enabling accurate classification on unseen data.

3.3.2 The Gradient Descent Update Rules

Gradient descent iteratively adjusts the parameters by moving in the direction of the negative gradient of J. Starting from some initial values, we repeat the following *simultaneous* updates until convergence:

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

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

where $\alpha > 0$ is the **learning rate**, a hyperparameter controlling the step size. Crucially, all parameters must be recomputed using the *old* values before any update is applied (simultaneous update).

3.3.3 Deriving the Gradients

Taking the partial derivatives of (3.4) and applying the chain rule, one can show:

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

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

Remark 3.3.1. The gradient expressions in (9.20) and (9.21) 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, $f = g(\vec{w} \cdot \vec{x} + b)$. This distinction makes the two algorithms fundamentally different, even though their gradient update forms look the same.

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

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

Averaging over all m examples gives (9.20). □

3.3.4 The Complete Algorithm

Algorithm: Gradient Descent for Logistic Regression

1. **Initialise** $\vec{w} = \mathbf{0}$ and $b = 0$ (or small random values).
2. **Repeat** until convergence:
   a. Compute $f^{(i)} = g(\vec{w} \cdot \vec{x}^{(i)} + b)$ for all i.
   b. Compute the gradients using (9.20) and (9.21).
   c. Simultaneously update: $w_j \leftarrow w_j - \alpha\, \partial J/\partial w_j$ and $b \leftarrow b - \alpha\, \partial J/\partial b$.
3. **Output** the trained parameters $(\vec{w}, b)$.

3.3.5 Practical Considerations

Choosing the learning rate α.. If α is too large, gradient descent may oscillate or diverge. If too small, convergence is slow. A practical strategy is to plot the *learning curve* — the cost J as a function of iteration count. A well-chosen α produces a monotonically decreasing, eventually flat, learning curve.

Feature scaling.. When features have vastly different magnitudes (e.g. $x_1 \in [1, 10^6]$ and $x_2 \in [0, 1]$), the cost surface becomes an elongated ellipse. Gradient descent then oscillates along the steep axis while making slow progress along the shallow one. Standardising features to have approximately equal scale (e.g. mean-zero, unit variance) transforms the contours into circles, allowing much faster convergence.

Vectorisation.. For large datasets with many features, computing the gradient using explicit `for`-loops is slow. Expressing the update as matrix–vector operations (e.g. using NumPy) allows the underlying linear algebra library to exploit parallelism and SIMD instructions, yielding orders-of-magnitude speedups.

Convergence criterion.. A common stopping rule is to halt when the relative change in J between successive iterations falls below a tolerance ε:

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

Alternatively, one may fix the number of iterations in advance based on the learning curve.

Scikit-learn.. In professional practice, logistic regression is usually trained using the highly optimised implementation in `scikit-learn` (`sklearn.linear_model.LogisticRegression`), which uses second-order methods (L-BFGS) and supports regularisation out of the box. Understanding the underlying mathematics, however, remains essential for debugging, model selection, and designing custom architectures.

3.4 The Problem of Overfitting

3.4.1 The Bias–Variance Trade-off

Every supervised learning algorithm must navigate a fundamental tension: a model that is too simple cannot capture the true structure of the data, while a model that is too complex memorises the training data and fails to generalise. This tension is formalsied by the **bias–variance trade-off**.

Underfitting (High Bias).. An underfitting model has a strong preconception (bias) about the form of the relationship. For example, 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, and there is little hope it will do better on unseen data.

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, achieving near-zero training error, but its highly oscillatory curve is driven by noise rather than the true signal. Small changes in the training set produce drastically different curves (high variance). The model fails to **generalise** to new data.

Good generalisation.. The ideal model captures the true underlying pattern while ignoring random noise. It achieves low error on both the training set and unseen test data. This is the goal of every machine learning pipeline.

3.4.2 Overfitting in Classification

In classification, overfitting manifests as a *decision boundary* that weaves intricately between training points, achieving perfect training accuracy at the cost of poor test accuracy. Adding high-degree polynomial features gives the model the flexibility to create such boundaries. Without a mechanism to control complexity, larger feature sets almost inevitably lead to overfitting when the training set is small.

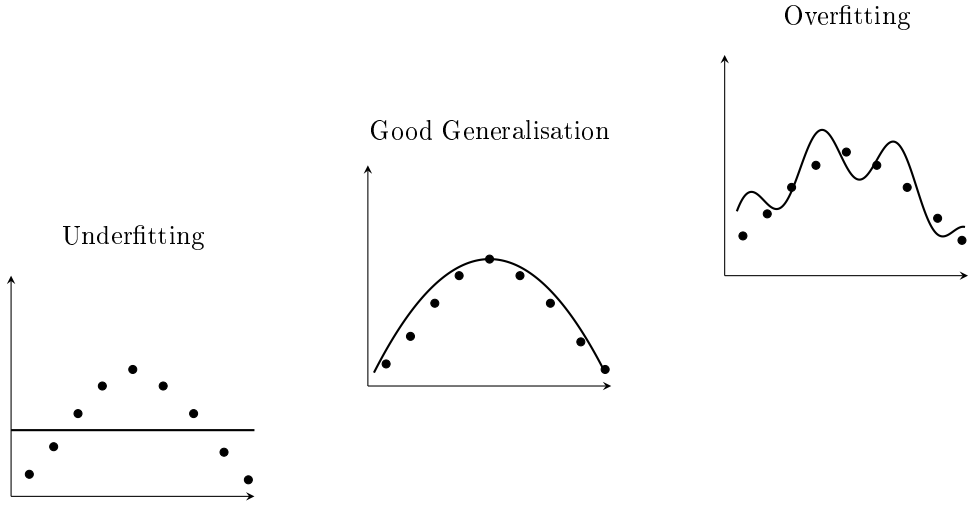


Figure 3.5. Three models fitted to the same data. *Left*: underfitting (too simple). *Centre*: good generalisation. *Right*: overfitting (too complex, "wiggly" curve).

3.4.3 Remedies for Overfitting

Strategy 1: Collect More Training Data

The most direct remedy is to increase the number of training examples m. 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 simply 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 or embedded methods).

The main drawback is that discarding features risks losing information that might have been genuinely predictive. Moreover, identifying the optimal subset is combinatorially hard for large n.

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 w_j are what allow a model to oscillate wildly. If we *penalise* large weights in the cost function, the optimiser is forced to keep them small, producing a smoother, more parsimonious model.

The regularised cost function (general form).. We add an L_2 (ridge) regularisation term to the original cost:

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

where $\lambda \geq 0$ is the **regularisation parameter** and J is the baseline cost (MSE for linear regression, cross-entropy for logistic regression). Note that b is conventionally *not* regularised.

The role of λ.. The scalar λ controls the trade-off between fitting the data and keeping the model simple:

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

3.4.4 Regularised Linear Regression

Cost function..

$$J(\vec{w}, b) = \underbrace{\frac{1}{2m} \sum_{i=1}^{m} (f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)})^2}_{*\text{MSE}} + \underbrace{\frac{\lambda}{2m} \sum *j = 1^{n} w_j^2}_{\text{regularisation}}. \tag{3.9}$$

Gradient descent update rules..

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

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

Weight decay interpretation.. Rearranging Equation (3.10):

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

Since α and λ are small positive numbers and $m \geq 1$, the weight decay factor is slightly less than 1 (e.g. 0.9998). At every step, each weight is *gently shrunk* before the gradient update, creating a constant restoring force toward zero. This is the mechanism by which regularisation prevents parameter explosion.

Calculus note.. The extra gradient term $\lambda w_j/m$ in Equation (3.10) arises from differentiating the regularisation term:

$$\frac{\partial}{\partial w_j}\left(\frac{\lambda}{2m}\sum_{k=1}^{n} w_k^2\right) = \frac{\lambda}{2m}\cdot 2w_j = \frac{\lambda}{m}\,w_j.$$

The factor $2m$ in the denominator of (3.9) is chosen precisely so that the 2 from the power-rule cancels, yielding the clean term $\lambda w_j/m$.

3.4.5 Regularised Logistic Regression

The same regularisation strategy applies directly to logistic regression.

Regularised cost function..

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

where $f^{(i)} = g(\vec{w}\cdot\vec{x}^{(i)} + b)$.

Gradient descent update rules..

$$w_j \leftarrow w_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}\big(f^{(i)} - y^{(i)}\big)x_j^{(i)} + \frac{\lambda}{m}w_j\right], \tag{3.13}$$

$$b \leftarrow b - \alpha\,\frac{1}{m}\sum_{i=1}^{m}\big(f^{(i)} - y^{(i)}\big). \tag{3.14}$$

Although Equations (3.13)–(3.14) are structurally identical to (3.10)–(3.11), the function f is the sigmoid in the logistic case. The two algorithms are therefore distinct, even though their update rules share the same symbolic form.

Remark 3.4.1. Regularisation smooths the decision boundary in classification by preventing the parameters of high-degree features from growing large. The result is a boundary that generalises better, correctly classifying unseen points rather than fitting every training point exactly.

3.4.6 Summary Table

Chapter Summary

This chapter introduced the fundamentals of binary classification through the lens of *logistic regression.*

Situation	Training Error	Test Error	Remedy
Underfitting (High Bias)	High	High	More features; more complex model
Good Generalisation	Low	Low	—
Overfitting (High Variance)	Very low	High	More data; fewer features; regularise

Table 3.1. Summary of the bias–variance trade-off and recommended remedies.

- **Logistic regression** uses the sigmoid function to map any real-valued linear combination of features onto the interval $(0, 1)$, yielding a probability estimate. The decision boundary arises where $\vec{w} \cdot \vec{x} + b = 0$ and can be made non-linear by including polynomial features.
- The **cross-entropy cost function** is derived from maximum likelihood estimation and is convex, guaranteeing that gradient descent converges to the global optimum. Squared error, by contrast, produces a non-convex surface with potential local minima.
- **Gradient descent** for logistic regression follows the same iterative update structure as for linear regression, but with the sigmoid function replacing the identity function. The same practical guidelines apply: choose α by monitoring the learning curve, scale features, and use vectorisation.
- **Overfitting** occurs when the model is too complex relative to the training data, leading to poor generalisation. The three main remedies are collecting more data, reducing the feature set, and **regularisation**. L_2 regularisation adds a penalty proportional to the sum of squared weights to the cost function, controlled by the hyperparameter λ.
- The same cost functions, gradient descent rules, and regularisation principles developed in this chapter form the foundational building blocks of modern **neural networks** and deep learning architectures covered in subsequent chapters.

CHAPTER 4

Neural Networks

Chapter Overview. Neural networks stack logistic-regression units into layers to learn hierarchical, nonlinear representations. This chapter covers the biological inspiration, feedforward architecture, layer-by-layer computation, TensorFlow implementation, and the mathematical foundations of vectorised forward propagation.

4.1 Neural Networks Intuition

4.1.1 Biological Inspiration

The original motivation behind **Artificial Neural Networks (ANNs)**, often simply called *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 on from strictly biological analogies, the foundational vocabulary and structural metaphor still draw 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 signal to downstream neurons.

An **artificial neuron** is a mathematical abstraction of this process. It takes numerical inputs, computes a weighted sum, and emits an *activation value* that serves as the input to neurons in the next layer. Figure 4.1 illustrates this parallel.

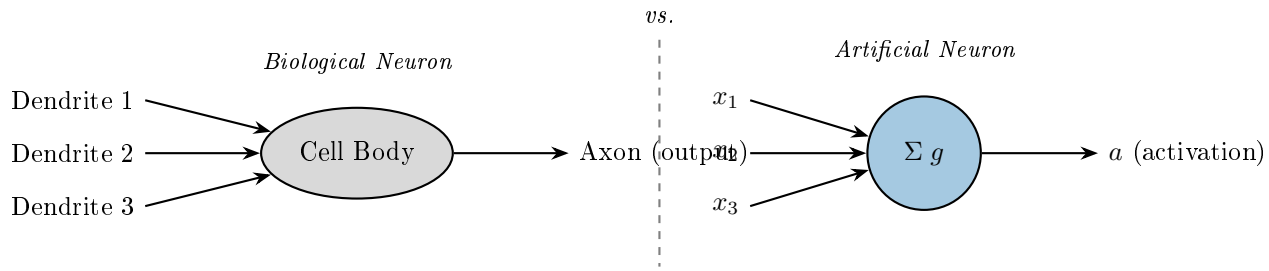


Figure 4.1. Structural analogy between a biological neuron and an artificial neuron.

Important caveat. 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 any faithful neurological simulation.

4.1.2 A Brief History

The development of neural networks has proceeded 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, checks)
Late 1990s	Decline	Outperformed by SVMs and other ML methods
2005–present	Resurgence	Rebranded as "Deep Learning"; dramatic breakthroughs

The most pivotal milestones in the modern era include:

- **Speech recognition** (early 2010s): Pioneered by Geoffrey Hinton and collaborators, deep learning first surpassed classical methods on speech tasks.
- **ImageNet 2012**: A landmark result in computer vision demonstrated that deep convolutional networks could recognise objects in images far better than prior approaches.
- **Natural Language Processing**: Following computer vision, deep learning transformed text understanding, culminating in Transformer-based large language models.
- **Today**: Neural networks underpin medical imaging, climate modelling, product recommendations, autonomous driving, and many other application domains.

4.1.3 Why Now? The Data and Compute Story

A natural question is: if these ideas existed for decades, why did they succeed only recently?

The plateau of classical machine learning. Traditional algorithms such as logistic regression or support vector machines exhibit a characteristic performance ceiling. Beyond a certain amount of training data, adding more examples yields diminishing returns and performance flattens, as depicted in Figure 4.2.

The neural network advantage. Large neural networks, by contrast, continued to improve as datasets grew. Two enabling factors catalysed this:

- **Explosion of digital data** generated by the internet and mobile devices.
- **GPU hardware**, which provides the massive parallel computation required to train large models efficiently.

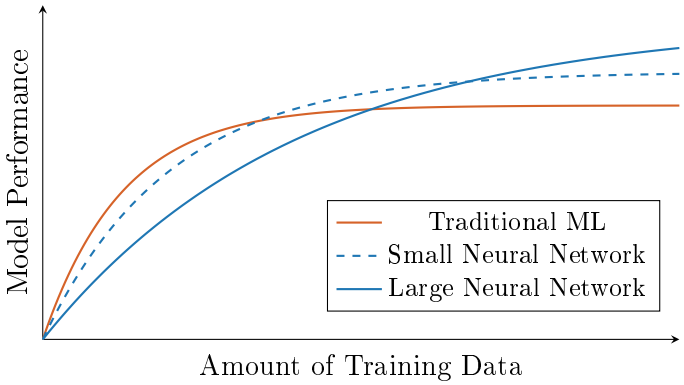


Figure 4.2. Schematic illustration of performance vs. training data for classical ML versus small and large neural networks. Large neural networks continue to benefit from additional data, whereas classical methods plateau.

4.2 Neural Network Model

4.2.1 The Single Neuron: Building Block

A single artificial neuron is the most elementary computational unit. Consider the task of predicting whether a T-shirt will become a top seller, given its price x.

The neuron applies a **sigmoid (logistic) activation function**:

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

where w is a learnable weight, b is a bias, and $a \in (0, 1)$ is the **activation**, representing the probability that the item is a top seller.

The terminology "activation" is borrowed from neuroscience: just as a biological neuron fires (activates) when stimulated strongly enough, an artificial neuron emits a large output when its weighted input crosses a threshold.

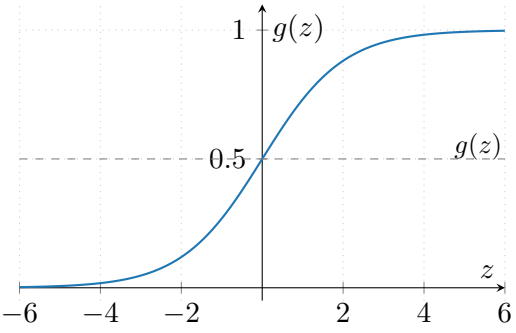


Figure 4.3. The sigmoid activation function $g(z) = 1/(1 + e^{-z})$.

4.2.2 Layered Architecture

When multiple neurons are combined in a *layered* structure, the network can capture complex nonlinear relationships. A standard feedforward architecture consists of:

- **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, typically a single probability for binary classification.

The term *hidden* refers to the fact that these layers are not directly observed in the training data: we see only the inputs $\vec{x}$ and the labels y, never the intermediate activations.

Figure 4.4 shows a concrete example for a demand-prediction problem with four input features.

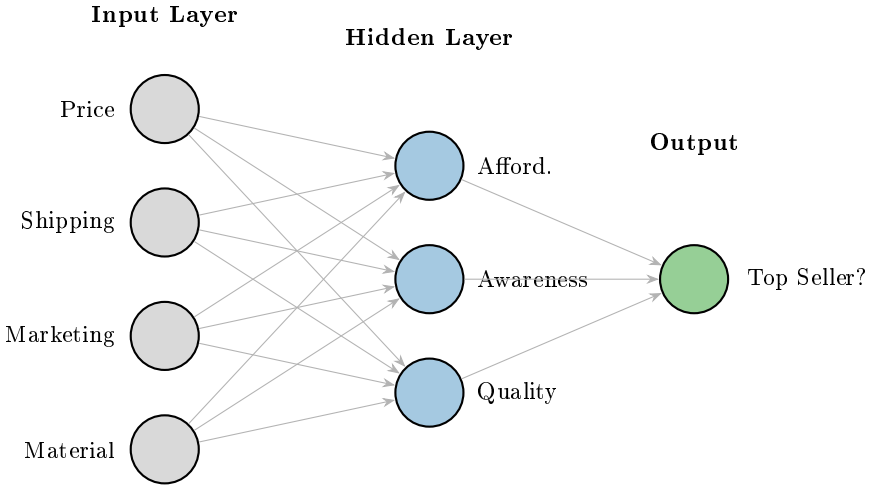


Figure 4.4. A neural network for demand prediction. Four input features connect to a hidden layer of three neurons, which in turn feed an output neuron.

4.2.3 Notation

A consistent notation is essential when working with multi-layer networks. We adopt the following conventions throughout this chapter:

- l in superscript square brackets denotes the **layer index**: $[1], [2], \ldots$
- j in subscript denotes the **neuron (unit) index** within a layer.
- $a_j^{[l]}$ is the activation of the j-th neuron in layer l.
- $\vec{w}_j^{[l]} \in \mathbb{R}^{n_{l-1}}$ and $b_j^{[l]} \in \mathbb{R}$ are the weight vector and bias of neuron j in layer l.
- $\vec{a}^{[l]}$ is the 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 any unit j, the activation is:

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

where g is the chosen activation function (sigmoid in this chapter). This single formula, applied repeatedly across layers and units, describes the entire computation of a feedforward neural network.

4.2.5 Layered Computation: A Detailed Example

Consider a network with 4 input features and layers of sizes 3, 3, and 1 (the architecture in Figure 4.4).

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

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

The layer output is the vector $\vec{a}^{[1]} = \left[a_1^{[1]}, a_2^{[1]}, a_3^{[1]}\right]^\top$.

Layer 2 (output, 1 neuron, input $\vec{a}^{[1]}$):

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

Final prediction: Given a threshold of 0.5,

$$\hat{y} = \begin{cases} 1 & \text{if } a_1^{[2]} \geq 0.5, \\ 0 & \text{otherwise.} \end{cases}$$

4.2.6 Multiple Hidden Layers and Deep Architectures

Networks with more than one hidden layer are referred to as **deep** networks, and the field studying them is called **deep learning**. A *4-layer network* means three hidden layers plus one output layer (the input layer is not counted):

$$\underbrace{\vec{x}}_{*\text{Layer 0}} \rightarrow \underbrace{\vec{a}^{[1]}}_{*\text{Layer 1}} \rightarrow \underbrace{\vec{a}^{[2]}}_{*\text{Layer 2}} \rightarrow \underbrace{\vec{a}^{[3]}}_{*\text{Layer 3}} \rightarrow \underbrace{\hat{y}}_{\text{Layer 4}}$$

The number of hidden layers and the number of units per layer are **architectural hyperparameters** that must be chosen by the practitioner. A network with multiple layers and varying unit counts is often called a **multi-layer perceptron (MLP)**.

4.2.7 Computer Vision: Hierarchical Feature Detection

Computer vision is one of the most instructive domains for building intuition about deep representations. An image is simply a matrix of pixel intensities (values from 0 to 255). For a 1000×1000 image, the input vector $\vec{x}$ has 10^6 components.

What do the hidden layers learn? Empirical visualisation studies reveal a remarkable hierarchy:

Layer	Features Detected	Description
1st hidden	Edges and oriented lines	Short line segments at various angles
2nd hidden	Object parts	Eyes, corners of noses, wheels, headlights
3rd hidden	Coarse shapes	Full face or car silhouettes
Output	Identity / category	Final classification probability

Crucially, **no human programmer defines these features**. The network discovers them automatically by minimising prediction error during training. Furthermore, the same architecture trained on a different dataset (e.g., cars instead of faces) will automatically learn domain-appropriate features in each layer.

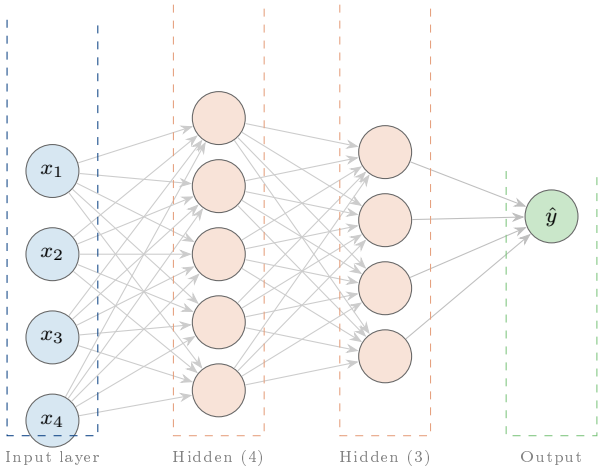


Figure 4.5. A fully connected feedforward neural network with four input features, two hidden layers, and one output unit. Each connection carries a learnable weight; every neuron also has a learnable bias. The network learns to approximate $\hat{y} = f(\mathbf{x})$ by adjusting all weights and biases during training.

4.3 TensorFlow Implementation

4.3.1 Data Representation in TensorFlow

TensorFlow processes data in **two-dimensional matrices** (called *tensors*) even for single samples. This design choice enables the framework to process large batches of examples efficiently using GPU parallelism.

Representation	NumPy Code	Shape
1D vector (scalar features)	`np.array([200, 17])`	$(2,)$ — no rows/columns
2D row vector (TF input)	`np.array([[200, 17]])`	1×2
2D column vector	`np.array([[200], [17]])`	2×1

The double-bracket notation `[[...]]` signals a 2D matrix. Always use 2D arrays when passing data to TensorFlow layers. A *Tensor* is TensorFlow's native data type — for our purposes, it behaves identically to a NumPy array. Tensors display their *shape* (e.g., `(1, 3)`) and *dtype* (typically `float32`).

Converting between frameworks is straightforward:

- NumPy → TensorFlow: happens automatically when you pass a NumPy array to a layer.
- TensorFlow → NumPy: call `tensor.numpy()`.

4.3.2 Building Layers: The Dense Layer

In TensorFlow/Keras, the standard fully-connected layer is called a **Dense** layer. Every unit in a Dense layer receives activations from *every* unit in the preceding layer — hence "dense" or "fully connected."

The two primary parameters are:

- `units`: number of neurons in the layer.
- `activation`: the activation function (e.g., `'sigmoid'`).

4.3.3 Example: Coffee Roasting Classification

Problem setup. Coffee is *good* ($y = 1$) only if temperature and duration remain within a specific triangular region — too hot/long burns it; too cool/short undercooks it. Inputs: Temperature (°C) and Duration (min).

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Dense

# Input features: temperature = 200 deg C, duration = 17 min

x = np.array([[200.0, 17.0]])    # shape (1, 2)

# Hidden layer: 3 neurons with sigmoid activation

layer_1 = Dense(units=3, activation='sigmoid')
a1 = layer_1(x)                  # shape (1, 3)

# Output layer: 1 neuron with sigmoid activation

layer_2 = Dense(units=1, activation='sigmoid')
a2 = layer_2(a1)                 # shape (1, 1)

# Binary prediction

yhat = 1 if a2.numpy()[0, 0] >= 0.5 else 0
```

Listing 4.1. Explicit layer-by-layer inference for coffee roasting.

4.3.4 The Sequential API

Instead of manually chaining layers, TensorFlow's **Sequential** model automates this by stringing layers into a single object.

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

# Define model architecture

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

Listing 4.2. Compact Sequential model definition.

4.3.5 The Machine Learning Workflow

After defining the architecture, training a model in TensorFlow follows three steps:

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

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

2. **Train** — fit the model to labelled data:

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

3. **Predict** — run forward propagation on new data:

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

4.3.6 Complete Coffee Roasting Example

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

# Training data: 4 samples, 2 features each

X = np.array([[200.0, 17.0],
[120.0,  5.0],
[425.0, 20.0],
[212.0, 18.0]])
y = np.array([1, 0, 0, 1])     # 1 = good, 0 = bad

# Model

```

```
model = Sequential([
Dense(units=3, activation='sigmoid'),
Dense(units=1, activation='sigmoid')
])

# Compile and train

model.compile(loss='binary_crossentropy', optimizer='adam')
model.fit(X, y, epochs=200, verbose=0)

# Inference on a new sample

X_new = np.array([[200.0, 13.9]])
prob = model.predict(X_new)
print(f"Probability of good coffee: {prob[0,0]:.4f}")
```

Listing 4.3. Full pipeline: data, model, training, and inference.

4.3.7 Digit Recognition Example

For a more complex task — distinguishing the handwritten digits 0 and 1 — we simply increase the depth and width of the network:

```
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 4.4. Neural network for handwritten digit recognition.

The input is a 1×64 row vector (an 8×8 pixel image flattened into 64 numbers). The network progressively compresses the representation from 64 dimensions down to a single probability.

4.4 Neural Network Implementation in Python

4.4.1 Forward Propagation from Scratch

Understanding how to implement forward propagation manually, using only NumPy, is essential for:

- **Debugging**: tracing errors in model outputs back to specific computations.
- **Insight**: appreciating the linear algebra that underlies every deep learning framework.
- **Custom architectures**: implementing non-standard layers when frameworks fall short.

In the scratch implementation, we use **1D NumPy arrays** (single brackets) rather than the 2D matrices required by TensorFlow. The mapping between mathematical notation and Python variables is:

Mathematical Symbol	Python Variable	Meaning
$\vec{w}_1^{[1]}$	`w1_1`	Weights of neuron 1, layer 1
$b_1^{[1]}$	`b1_1`	Bias of neuron 1, layer 1
$a_1^{[1]}$	`a1_1`	Activation of neuron 1, layer 1

4.4.2 Computing a Layer Neuron-by-Neuron

The sigmoid function is defined as:

```
import numpy as np

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

Listing 4.5. Sigmoid helper function.

For the coffee roasting model, Layer 1 has 3 neurons receiving a 2-dimensional input x = [temp, duration]:

```
x = np.array([200.0, 17.0])    # Input features

# Neuron 1

w1_1 = np.array([1.0, 2.0]);  b1_1 = -1.0
z1_1  = np.dot(w1_1, x) + b1_1
a1_1  = sigmoid(z1_1)

# Neuron 2

w1_2 = np.array([-3.0, 4.0]); b1_2 =  2.0
z1_2  = np.dot(w1_2, x) + b1_2
a1_2  = sigmoid(z1_2)

# Neuron 3

w1_3 = np.array([5.0, -6.0]); b1_3 = -3.0
z1_3  = np.dot(w1_3, x) + b1_3
a1_3  = sigmoid(z1_3)

# Collect activations into a vector

a1 = np.array([a1_1, a1_2, a1_3])
```

Listing 4.6. Neuron-by-neuron forward pass for Layer 1.

Layer 2 (the output neuron) then receives $\vec{a}^{[1]}$:

```
w2_1 = np.array([-7.0, 8.0, 9.0]); b2_1 = 3.0
z2_1  = np.dot(w2_1, a1) + b2_1
```

```
a2_1  = sigmoid(z2_1)    # Final probability
```

Listing 4.7. Forward pass for the output layer.

The core mathematical operation at every neuron is a **dot product** followed by a bias addition and a nonlinear function:

$$a_j^{[l]} = g\left(\sum_{i=1}^{n} w_{ji}^{[l]}\, a_i^{[l-1]} + b_j^{[l]}\right).$$

4.4.3 The `dense()` Function

To avoid writing one block of code per neuron, we generalise using matrix notation. We stack each neuron's weight vector as a *column* of a weight matrix W:

$$W^{[l]} = \left[\vec{w}_1^{[l]} \,\middle|\, \vec{w}_2^{[l]} \,\middle|\, \cdots \,\middle|\, \vec{w}_{n_l}^{[l]}\right] \in \mathbb{R}^{n_{l-1}\times n_l},$$

and collect the biases into $\vec{b}^{[l]} \in \mathbb{R}^{n_l}$.

```
def dense(a_in, W, b, activation=sigmoid):
"""
Compute activations for one dense layer.

Parameters
----------
a_in : ndarray, shape (n_in,)
    Activations from the previous layer.
W    : ndarray, shape (n_in, n_units)
    Weight matrix; each COLUMN belongs to one neuron.
b    : ndarray, shape (n_units,)
    Bias vector.

Returns
-------
a_out : ndarray, shape (n_units,)
    Activations of this layer.
"""
n_units = W.shape[1]
a_out   = np.zeros(n_units)

for j in range(n_units):
    w_j       = W[:, j]                # j-th column = weights of neuron j
    z_j       = np.dot(w_j, a_in) + b[j]
    a_out[j] = activation(z_j)

return a_out
```

Listing 4.8. Reusable `dense()` function for a single fully-connected layer.

The slice notation `W[:, j]` extracts all rows of column j, i.e., the weight vector for neuron j.

4.4.4 The `sequential()` Function

Once `dense()` is available, stringing multiple layers together is straightforward:

```
def sequential(x, W1, b1, W2, b2, W3, b3, W4, b4):
"""Full forward pass through a 4-layer network."""
a1   = dense(x,  W1, b1)
a2   = dense(a1, W2, b2)
a3   = dense(a2, W3, b3)
a4   = dense(a3, W4, b4)
f_x  = a4
return f_x
```

Listing 4.9. Manual sequential forward propagation for four layers.

4.4.5 Forward Propagation: The Handwritten Digit Case

For the 8×8 digit recognition task, the input is a 64-element vector and the network architecture is $64 \to 25 \to 15 \to 1$:

```

# Assume W1, b1, W2, b2, W3, b3 are pre-trained weight arrays

# x : shape (64,)

a1 = dense(x,  W1, b1)   # (25,) -- 25 units in Layer 1
a2 = dense(a1, W2, b2)   # (15,) -- 15 units in Layer 2
a3 = dense(a2, W3, b3)   # (1,)  -- 1 unit  in Layer 3

yhat = 1 if a3[0] >= 0.5 else 0
print(f"Predicted digit: {yhat}")
```

Listing 4.10. Forward propagation steps for digit recognition.

4.5 Speculations on Artificial General Intelligence (AGI)

4.5.1 From Narrow AI to AGI

Current neural networks are examples of **Artificial Narrow Intelligence (ANI)**: they are trained for a specific task and perform well on that task — and only on that task. Examples include:

- A speech-to-text model that cannot play chess.
- An image classifier that cannot write poetry.
- A recommendation engine that cannot navigate a car.

Artificial General Intelligence (AGI), by contrast, would be a system capable of *any intellectual task that a human can perform.* AGI remains one of the central unsolved problems in computer science and cognitive science.

4.5.2 The Biological Plausibility Gap

One recurring speculation is that AGI may require fundamentally new approaches, because modern deep learning is quite unlike the brain:

- **Learning efficiency**: a child learns to recognise a dog from a handful of examples; a deep network may require millions.
- **Generalisation**: humans transfer knowledge across domains fluidly; neural networks generalise poorly outside their training distribution.
- **Causal reasoning**: humans model cause and effect; current networks primarily detect statistical correlations.
- **Energy consumption**: the human brain operates on roughly 20 watts; training a large language model consumes megawatt-hours.

4.5.3 The Optimistic View

Some researchers argue that scaling existing architectures — more parameters, more data, more compute — may be sufficient to approach AGI. Evidence includes:

- Emergent capabilities in large language models that were not explicitly trained (few-shot learning, chain-of-thought reasoning).
- The consistent historical pattern that increasing data and compute has reliably improved performance on almost every benchmark.

4.5.4 The Cautious View

Others contend that current deep learning is fundamentally limited:

- Gradient-based learning in fixed-topology networks may be insufficient for the open-ended, cumulative learning seen in biological intelligence.
- Without grounded world models, purely pattern-matching systems cannot achieve robust generalisation.

4.5.5 Timeline and Uncertainty

Credible estimates for AGI range from ‘within the decade” to‘ never, as currently conceived.” This wide disagreement among experts highlights how poorly we understand both the requirements for general intelligence and the trajectory of progress in AI research.

For the practitioner, the more immediate question is not when AGI arrives, but whether the tools and techniques studied in this chapter will play a role in that trajectory — and the answer appears to be: very likely, yes.

4.6 Vectorization

4.6.1 Motivation: The Cost of Python Loops

The `dense()` function introduced in Section IV computes each neuron's activation in a `for` loop. For a layer with n units receiving m inputs, this requires n separate dot-product calls. In Python, interpreted loops are slow; for networks with thousands of units and millions of training examples, this becomes the bottleneck.

The solution is **vectorization**: replacing explicit loops with single matrix operations that can be executed by highly optimised hardware (CPUs with SIMD, or GPUs).

4.6.2 The Matrix Formulation

Instead of computing each neuron separately, we treat the entire layer as a single linear transformation followed by an element-wise nonlinearity.

Let:

- $\vec{A} * in$ be a **row vector** of shape $1 \times n * l - 1$ (one input sample, n_{l-1} features).
- $W^{[l]}$ be the weight matrix of shape $n_{l-1} \times n_l$ (each column = one neuron's weights).
- $\vec{B}^{[l]}$ be the bias row vector of shape $1 \times n_l$.

Then the entire layer computes:

$$Z = \vec{A} * in\, W^{[l]} + \vec{B}^{[l]}, \qquad \vec{A} * out = g(Z), \tag{4.3}$$

where g is applied *element-wise* to every entry of Z.

4.6.3 Dimension Alignment

For equation (4.3) to be valid, the matrix dimensions must be compatible:

Quantity	Variable	Shape	Interpretation
Input activations	$\vec{A} * in$	$1 \times n * l - 1$	1 sample, n_{l-1} features
Weight matrix	$W^{[l]}$	$n_{l-1} \times n_l$	n_{l-1} inputs, n_l neurons
Bias vector	$\vec{B}^{[l]}$	$1 \times n_l$	one bias per neuron
Output activations	$\vec{A}_{out}$	$1 \times n_l$	one activation per neuron

The matrix product $\vec{A} * inW^{[l]}$ has shape $(1 \times n * l - 1) \times (n_{l-1} \times n_l) = 1 \times n_l$, which matches $\vec{B}^{[l]}$. Adding the bias and applying g element-wise yields $\vec{A}_{out}$ of shape $1 \times n_l$.

4.6.4 Vectorized Dense Layer

```
def dense_vectorized(A_in, W, b, activation=sigmoid):
"""
Vectorized forward pass for one dense layer.

Parameters
----------
A_in : ndarray, shape (1, n_in)  -- 2D row vector
W    : ndarray, shape (n_in, n_units)
b    : ndarray, shape (1, n_units)  or broadcastable

Returns
-------
A_out : ndarray, shape (1, n_units)
"""
Z     = np.matmul(A_in, W) + b    # (1, n_in) @ (n_in, n_units) = (1,
    n_units)
A_out = activation(Z)              # element-wise sigmoid
return A_out
```

Listing 4.11. Vectorized `dense()` function using `np.matmul`.

4.6.5 Side-by-Side Comparison

```

# ---- Iterative (slow) -----

def dense_loop(a_in, W, b):
n_units = W.shape[1]
a_out   = np.zeros(n_units)
for j in range(n_units):
z        = np.dot(W[:, j], a_in) + b[j]
a_out[j] = sigmoid(z)
return a_out

# ---- Vectorized (fast) -----

def dense_vec(A_in, W, b):
Z     = np.matmul(A_in, W) + b    # Single matrix-multiply
A_out = sigmoid(Z)
return A_out
```

Listing 4.12. Iterative vs. vectorized implementation.

The vectorized version replaces n_l individual dot products with a single call to `np.matmul`, which internally delegates to BLAS/LAPACK routines or GPU kernels.

4.6.6 Batch Processing

The formulation naturally extends to processing a **batch** of m examples simultaneously. If $\vec{A} * in$ has shape $m \times n * l - 1$ (one row per sample), then:

$$Z = \vec{A}_{in} W^{[l]} + \vec{B}^{[l]}, \qquad Z \in \mathbb{R}^{m \times n_l},$$

computing all m samples in one matrix multiplication. This is exactly the design philosophy of TensorFlow and PyTorch, and the reason both frameworks require 2D (or higher-dimensional) input tensors.

4.6.7 Why SIMD / GPU Matters

Modern CPUs and GPUs implement **Single Instruction Multiple Data (SIMD)** pipelines. A single instruction can multiply-accumulate entire vectors of floating-point numbers simultaneously. For a matrix multiplication of shape $(m \times k) \times (k \times n)$, a GPU can compute all mn output elements nearly in parallel.

Figure 4.6 schematically illustrates the difference between serial and vectorized computation.

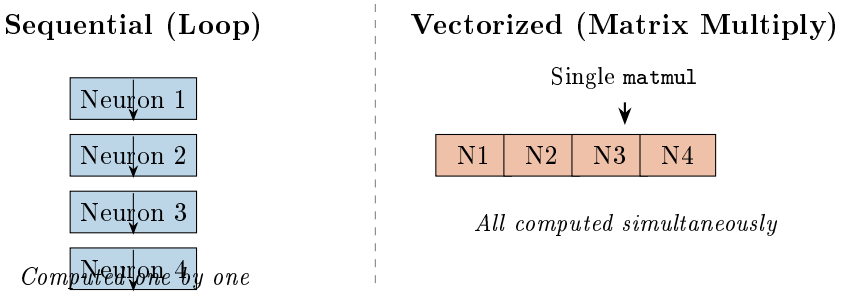


Figure 4.6. Sequential (loop-based) vs. vectorized (matrix-multiply) computation of a neural network layer.

4.6.8 Summary of the Vectorization Equations

For a complete network, vectorized forward propagation proceeds as follows:

$$\begin{aligned}
\vec{A}^{[0]} &= X, \\
Z^{[l]} &= \vec{A}^{[l-1]} W^{[l]} + \vec{B}^{[l]}, \quad l = 1, 2, \ldots, L, \\
\vec{A}^{[l]} &= g\Big(Z^{[l]}\Big), \\
\hat{Y} &= \vec{A}^{[L]}.
\end{aligned}$$

Each line is a single NumPy or TensorFlow operation; no Python loop over neurons or layers is required beyond the L layer steps shown above.

CHAPTER 5

Neural Network Training

Chapter Overview. This chapter covers the full training pipeline for neural networks: the three-step compile/fit framework, activation function choices (ReLU vs. sigmoid), multiclass classification with softmax, the critical `from_logits` pattern for numerical stability, multi-label classification, and the backpropagation algorithm that makes efficient gradient computation possible.

5.1 Neural Network Training

5.1.1 From Inference to Training

Up to this point we have used neural networks in *inference* mode: given a fixed, pre-trained set of parameters $\{\mathbf{W}^{[l]}, \mathbf{b}^{[l]}\}$ we feed an input $\vec{x}$ through the network and obtain a prediction $\hat{y}$. Training is the complementary process: given a labelled dataset $\{(\vec{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$, we *learn* the parameters that minimise a chosen measure of error.

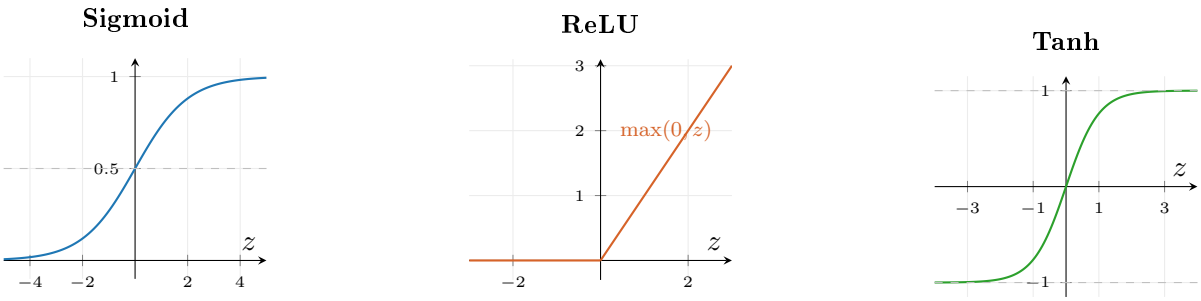


Figure 5.1. Three common activation functions. **Sigmoid** squashes output to $(0, 1)$, suitable for binary output nodes. **ReLU** (Rectified Linear Unit) outputs $\max(0, z)$—sparse, cheap to compute, and the standard choice for hidden layers. **Tanh** outputs values in $(-1, 1)$, zero-centred, often preferred over sigmoid in hidden layers.

In this chapter we use a running case study—binary classification of handwritten digits (‘0” vs. ‘ 1”)—to ground the discussion before extending to richer settings.

5.1.2 A Three-Step Training Framework

Whether we are 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 $\vec{x} \to \hat{y}$	$f(x) = \sigma(\vec{w} \cdot \vec{x} + b)$	`Sequential([Dense(...)])`
2	Measure error	Logistic loss $J(\vec{w}, b)$	`model.compile(loss=...)`
3	Minimise error	Gradient descent	`model.fit(X,y,epochs=...)`

Step 1 — Specify the Architecture

The network for our binary digit classifier consists of three dense layers:

Layer	Type	Units	Activation
1	Hidden	25	Sigmoid
2	Hidden	15	Sigmoid
3	Output	1	Sigmoid

```
import tensorflow as tf
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. Step 1 — defining the model architecture in TensorFlow.

TensorFlow automatically initialises the weight matrices $\mathbf{W}^{[l]}$ and bias vectors $\mathbf{b}^{[l]}$ for every layer.

Step 2 — Compile: Specify the Loss Function

The loss function quantifies how far the model's prediction $\hat{y} = f(\vec{x})$ deviates from the true label y.

Binary Cross-Entropy Loss. For a single training example $(\vec{x}, y)$ with $y \in \{0, 1\}$:

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

The *cost function* averages the loss over all m training 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())
```

Listing 5.2. Step 2 — compiling the model with binary cross-entropy loss.

Mean Squared Error Loss (Regression). When predicting a continuous numerical output, replace the cross-entropy with the squared-error loss:

$$L\big(f(\vec{x}),\, y\big) = \tfrac{1}{2}\big(f(\vec{x}) - y\big)^2. \tag{5.3}$$

```
from tensorflow.keras.losses import MeanSquaredError

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

Listing 5.3. Compiling with mean squared error for regression tasks.

Step 3 — Train: Gradient Descent and Backpropagation

To minimise $J(\mathbf{W}, \mathbf{B})$ we update every parameter in every layer l iteratively:

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

$$b_j^{[l]} \leftarrow b_j^{[l]} - \alpha \frac{\partial J}{\partial b_j^{[l]}}. \tag{5.5}$$

Here $\alpha > 0$ is the *learning rate*. Computing the partial derivatives in (5.4)–(5.5) across all layers is the job of **backpropagation** (detailed in Section 5.5).

In TensorFlow, a single call to `model.fit()` executes the full training loop: it runs forward propagation, computes gradients via backpropagation, applies the gradient-descent update, and repeats for the specified number of *epochs*.

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

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

Listing 5.4. Step 3 — fitting the model for 100 epochs.

5.1.3 Conceptual Importance of the Underlying Mathematics

A common pitfall for practitioners is to treat the three-step framework as a "black box": define a `Sequential` model, call `compile`, call `fit`, and hope for the best. While this workflow is correct, it does not equip you to *debug* a model that fails to learn.

Understanding the mathematical foundations—the loss function, the gradient, and the update rule—allows you to diagnose problems such as:

- a loss that fails to decrease (bad learning rate or wrong activation);
- a loss that oscillates wildly (learning rate too large);
- a model that learns on training data but generalises poorly (overfitting).

Throughout this chapter we therefore balance *implementation* fluency with *conceptual* understanding.

5.2 Activation Functions

5.2.1 Motivation: Why Not Just Use Sigmoid Everywhere?

Early neural networks were built by stacking *logistic regression* units, each using the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ as its activation. While sigmoid is well-suited to binary output nodes, it has two important limitations when used in *hidden* layers:

1. **Range constraint.** Sigmoid is strictly bounded in $(0, 1)$. Hidden-layer activations that represent quantities such as 'awareness" or' demand" may naturally take arbitrarily large non-negative values; clamping them to $(0, 1)$ discards information.
2. **Vanishing gradients.** At both ends of the sigmoid curve the derivative approaches zero. During backpropagation, this causes gradients to shrink exponentially as they are multiplied layer by layer, severely slowing down learning in deep networks.

5.2.2 A Catalogue of Common Activation Functions

Sigmoid

$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}, \qquad g(z) \in (0, 1). \tag{5.6}$$

The output can be interpreted as $P(y = 1 \mid \vec{x})$, making sigmoid the natural choice for the output neuron of a *binary classifier*.

ReLU — Rectified Linear Unit

$$g(z) = \max(0,\, z), \qquad g(z) \in [0, \infty). \tag{5.7}$$

ReLU is the **default activation for hidden layers** in modern deep learning. Its piecewise-linear form gives it two key advantages:

- **Computational efficiency.** Evaluating $\max(0, z)$ is a single comparison, far cheaper than the exponentiation required by sigmoid.
- **Non-vanishing gradients.** For $z > 0$ the gradient is exactly 1, so gradients flow backwards through the network without shrinking. Only for $z < 0$ is the gradient zero (the "dead ReLU" region).

Linear (Identity)

$$g(z) = z, \qquad g(z) \in (-\infty, +\infty). \tag{5.8}$$

Practitioners often call this "no activation function." It is appropriate *only* in the output layer of a regression model whose target y can take any real value.

Visual Comparison

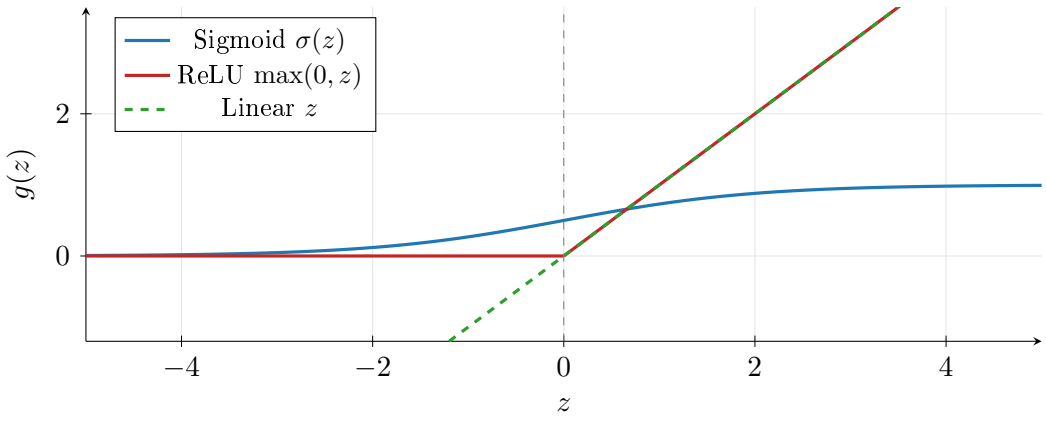


Figure 5.2. Comparison of Sigmoid, ReLU, and Linear activation functions. Note how Sigmoid saturates at both ends, while ReLU maintains a constant gradient of 1 for all positive inputs.

5.2.3 Choosing the Right Activation Function

Output Layer: Determined by the Task

Task	**Target y**	**Activation**	**Output 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 the Standard Default

Property	**Sigmoid**	**ReLU**
Gradient for large $\lvert z \rvert$	≈ 0 (vanishing)	1 (constant for $z > 0$)
Computational cost	High (exp, division)	Low (max operation)
Risk of vanishing gradients	High	Low

The **practical recommendation**: use ReLU in all hidden layers and select the output activation based on the task type.

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

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

Listing 5.5. Recommended architecture: ReLU hidden layers with sigmoid output.

Note. Alternative activation functions exist—Tanh, Leaky ReLU, Swish, GELU—and can outperform ReLU in specific architectures (transformers, generative models). Nevertheless, ReLU remains the recommended starting point for the vast majority of feedforward networks.

5.2.4 The Necessity of Non-Linearity

The Linear Collapse Problem

A subtle but critical question: what happens if we use *linear* activations in every hidden layer? The answer is that the network collapses into a simple linear model, regardless of its depth.

Mathematical Proof

Consider the simplest possible network: one input x, one hidden unit with a linear activation, and one output unit.

Forward pass..

$$a^{[1]} = g(w_1 x + b_1) = w_1 x + b_1, \tag{5.9}$$
$$a^{[2]} = g(w_2 a^{[1]} + b_2) = w_2 a^{[1]} + b_2. \tag{5.10}$$

Substitution..

$$\begin{aligned} a^{[2]} &= w_2(w_1 x + b_1) + b_2 \\ &= (w_2 w_1)\, x + (w_2 b_1 + b_2). \end{aligned} \tag{5.11}$$

Setting $W = w_2 w_1$ and $B = w_2 b_1 + b_2$ gives

$$a^{[2]} = Wx + B, \tag{5.12}$$

which is exactly *linear regression*. The hidden layer contributes nothing beyond what a single linear unit could achieve.

Generalisation to Deep Networks

This argument extends inductively to any depth:

- **All layers linear** $\Rightarrow$ equivalent to linear regression (can only fit straight lines/flat hyperplanes).
- **Hidden layers linear, output sigmoid** $\Rightarrow$ equivalent to logistic regression (cannot learn non-linear decision boundaries).

Remark 5.2.1. Adding more *linear* layers provides *zero* additional expressive power. Non-linear activation functions—ReLU being the canonical choice—are what allow deep networks to learn hierarchical, complex features.

Practical rule of thumb..

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

5.3 Multiclass Classification

5.3.1 Definition and Motivation

Definition 5.3.1 (Multiclass Classification)**.** A supervised learning problem is called **multiclass** when the target variable y can take on more than two discrete values: $y \in \{1, 2, \ldots, N\}$ for some $N > 2$.

Example 5.3.1 (Handwritten Digit Recognition (MNIST))**.** Given an image of a handwritten digit, classify it as one of 10 possible classes: $y \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Other real-world instances include:

- **Medical diagnosis**: classify a patient's condition as one of several diseases (Disease A, B, C, or Healthy).
- **Manufacturing quality control**: identify defect type (scratch, discoloration, chip, or no defect).
- **Natural language processing**: part-of-speech tagging, intent classification.

5.3.2 Binary vs. Multiclass: A Comparison

Feature	Binary	Multiclass
Output y	Single scalar $\in \{0, 1\}$	Single integer $\in \{1, \ldots, N\}$
Probability estimated	$P(y = 1 \mid \vec{x})$	$P(y = j \mid \vec{x})$ for all j
Decision boundaries	One boundary	Multiple boundaries
Output activation	Sigmoid	Softmax
Loss function	Binary cross-entropy	Categorical cross-entropy

5.3.3 Softmax Regression

Generalising Logistic Regression

Logistic regression can be understood as a special case of *softmax regression* with $N = 2$:

$$z = \vec{w} \cdot \vec{x} + b, \tag{5.13}$$

$$a_1 = \sigma(z) = P(y = 1 \mid \vec{x}), \tag{5.14}$$

$$a_2 = 1 - a_1 = P(y = 0 \mid \vec{x}), \qquad a_1 + a_2 = 1. \tag{5.15}$$

Softmax generalises this to N classes by maintaining a dedicated parameter vector $(\vec{w}_j, b_j)$ for each class j.

Mathematical Formulation

Step 1: Linear terms..

$$z_j = \vec{w}_j \cdot \vec{x} + b_j, \qquad j = 1, \ldots, N. \tag{5.16}$$

Step 2: Softmax activation..

$$\boxed{a_j = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}} = P(y = j \mid \vec{x}).} \tag{5.17}$$

Key property (normalisation)..

$$\sum_{j=1}^{N} a_j = 1. \tag{5.18}$$

This ensures that $\mathbf{a} = (a_1, \ldots, a_N)$ forms a valid probability distribution over the N classes.

Cross-Entropy Loss for Softmax

Per-example Loss. If the ground-truth label for example i is $y^{(i)} = j$, then the loss is:

$$L(a_1, \ldots, a_N, y) = -\log(a_j) \quad \text{if } y = j. \tag{5.19}$$

Intuition.

- If $a_j \to 1$ (model is confident and correct): $-\log(a_j) \to 0$ (low loss).
- If $a_j \to 0$ (model is confident and wrong): $-\log(a_j) \to \infty$ (high loss).

The Total Cost Function.

$$J(\mathbf{W}, \mathbf{B}) = \frac{1}{m} \sum_{i=1}^{m} L\big(f(\vec{x}^{(i)}), y^{(i)}\big). \tag{5.20}$$

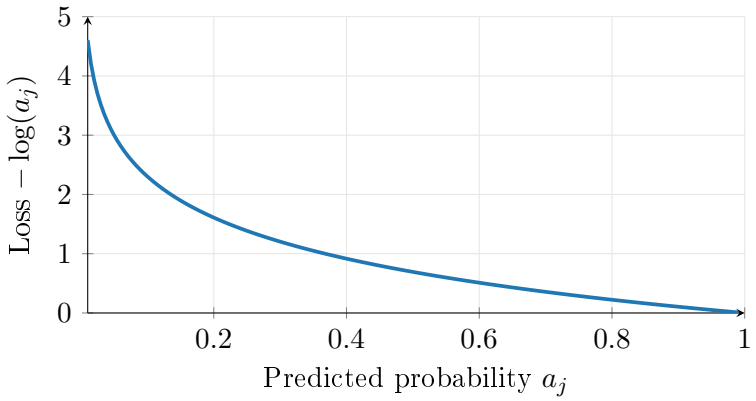


Figure 5.3. Cross-entropy loss $-\log(a_j)$ as a function of the predicted probability a_j for the correct class. The loss is zero only when the model is perfectly confident, and grows without bound as confidence in the correct class approaches zero.

5.3.4 Neural Network Architecture for Multiclass Classification

Structural Changes

To classify into N categories, the output layer must contain exactly N neurons and use the softmax activation:

Forward Propagation in the Softmax Layer

For the output layer (Layer 3, $j \in \{1, \ldots, 10\}$):

Linear terms..

$$z_j^{[3]} = \vec{w}_j^{[3]} \cdot \vec{a}^{[2]} + b_j^{[3]}. \tag{5.21}$$

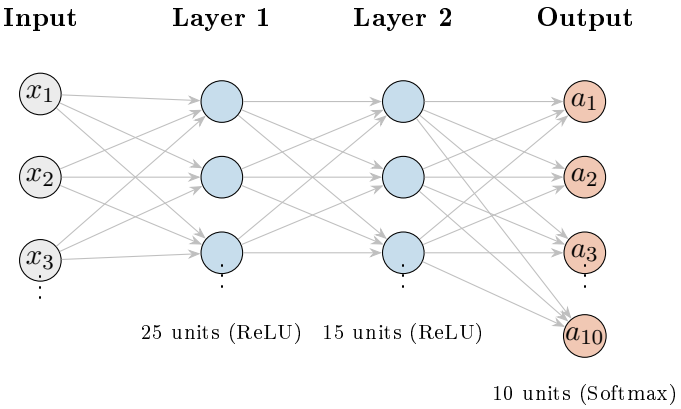


Figure 5.4. Neural network with softmax output for 10-class digit recognition. Each output neuron a_j represents $P(y = j \mid \vec{x})$, and the ten outputs sum to 1.

Softmax activation (non-element-wise)..

$$a_j^{[3]} = \frac{e^{z_j^{[3]}}}{\sum_{k=1}^{10} e^{z_k^{[3]}}} = P(y = j \mid \vec{x}). \tag{5.22}$$

Remark 5.3.1 (Coupling across units)**.** Unlike Sigmoid or ReLU, Softmax is **not element-wise**. Computing a_1 requires all ten z values simultaneously because of the shared normalisation denominator.

TensorFlow Implementation

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.losses import SparseCategoricalCrossentropy

model = Sequential([
Dense(units=25, activation='relu'),      # Hidden layer 1
Dense(units=15, activation='relu'),      # Hidden layer 2
Dense(units=10, activation='softmax'),   # Softmax output
])

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

Listing 5.6. Softmax network for MNIST digit recognition.

Loss function terminology..

- **Categorical**: the target is a discrete category label.

- **Sparse**: each training label is a single integer (e.g. $y = 7$), rather than a one-hot encoded vector of length 10.

5.4 Additional Neural Network Concepts

5.4.1 Numerical Stability: The `from_logits` Pattern

The Problem with Floating-Point Precision

When TensorFlow evaluates the softmax followed by the log inside the cross-entropy loss, it computes an intermediate probability a_j and then evaluates $-\log(a_j)$. For very large or very small values of z, this two-step computation suffers from *floating-point roundoff errors*:

- **Overflow**: e^z becomes too large to represent in memory.
- **Underflow**: e^z rounds to zero, making $\log(a_j) = -\infty$.

The Numerically Stable Solution

Instead of having the final layer output probabilities, we let it output the raw **logits** (the z values) and pass `from_logits=True` to the loss function. TensorFlow then combines the softmax and the log into a single, algebraically rearranged formula that avoids catastrophically small or large intermediate values.

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

Binary Classification (Numerically Stable).

```
from tensorflow.keras.losses import BinaryCrossentropy

model = Sequential([
Dense(units=25, activation='relu'),
Dense(units=15, activation='relu'),
Dense(units=1,  activation='linear'),  # Output logits, not probabilities
])

model.compile(loss=BinaryCrossentropy(from_logits=True))
model.fit(X, Y, epochs=100)
```

Listing 5.7. Numerically stable binary classification model.

Multiclass (Numerically Stable — Recommended Version).

```
from tensorflow.keras.losses import SparseCategoricalCrossentropy

model = Sequential([
Dense(units=25, activation='relu'),
Dense(units=15, activation='relu'),
Dense(units=10, activation='linear'),  # Output logits!
```

```
])

model.compile(loss=SparseCategoricalCrossentropy(from_logits=True))
model.fit(X, Y, epochs=100)
```

Listing 5.8. Numerically stable softmax model — the recommended production version.

Obtaining Probabilities at Inference Time

Because the model now outputs logits, you must explicitly apply the activation function when predicting:

```
import tensorflow as tf

logits = model.predict(X)

# For binary classification

probabilities = tf.nn.sigmoid(logits)

# For multiclass classification

probabilities = tf.nn.softmax(logits)
```

Listing 5.9. Converting logits to probabilities at inference time.

5.4.2 Multi-Label Classification

Definition

Definition 5.4.2 (Multi-label Classification)**.** In **multi-label** classification the model must predict a *vector* of binary labels for a single input. Multiple labels can be simultaneously active (non-mutually exclusive).

Example 5.4.1 (Autonomous Driving Perception)**.** Given a camera image, predict which objects are present:

$$y = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \begin{matrix} \leftarrow \text{Car present} \\ \leftarrow \text{No bus} \\ \leftarrow \text{Pedestrian present} \end{matrix}$$

Here the car and pedestrian can be present simultaneously.

Multi-Label vs. Multi-Class

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

Architecture

The recommended architecture uses **one network with N sigmoid outputs**:

```
model = Sequential([
Dense(units=25, activation='relu'),
Dense(units=15, activation='relu'),
# N independent sigmoid outputs, one per label
Dense(units=3,  activation='sigmoid'),
])

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

Listing 5.10. Multi-label classification with a single shared network.

Each output neuron independently predicts $P(\text{label}_j = 1 \mid \vec{x})$. This is equivalent to running N separate binary classifiers that share the same hidden-layer feature representations.

Remark 5.4.1. The loss is **Binary** Cross-Entropy (not Categorical), because each output neuron solves an independent binary prediction problem.

5.4.3 The Adam Optimiser

Limitations of Fixed-Learning-Rate Gradient Descent

Standard gradient descent uses a single, fixed learning rate α for all parameters. This leads to two failure modes:

- α **too small**: Learning is slow. Even if the gradient consistently points in one direction, progress is unnecessarily incremental.
- α **too large**: The update overshoots the minimum, causing oscillations that prevent convergence.

Adaptive Moment Estimation (Adam)

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

The adaptive intuition is straightforward:

- If parameter w_j consistently updates in the *same direction*, α_j is *increased* (the learning rate was too conservative).
- If parameter w_j *oscillates* in direction, α_j is *decreased* (the learning rate was too aggressive).

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

The update rules for n parameters are:

$$w_j \leftarrow w_j - \alpha_j \frac{\partial J}{\partial w_j}, \qquad j = 1, \ldots, n. \tag{5.23}$$

where each α_j is adapted separately on the basis of gradient history.

TensorFlow Implementation

```
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import SparseCategoricalCrossentropy

model = Sequential([
Dense(units=25, activation='sigmoid'),
Dense(units=15, activation='sigmoid'),
Dense(units=10, activation='linear'),
])

model.compile(
optimizer=Adam(learning_rate=1e-3),          # Initial learning rate
loss=SparseCategoricalCrossentropy(from_logits=True),
)

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

Listing 5.11. Using Adam instead of vanilla gradient descent in TensorFlow.

The `learning_rate` argument provides the *initial* global learning rate; Adam subsequently adjusts individual rates per parameter. Common starting values are 10^{-3}, 10^{-4}, or 10^{-5}.

5.4.4 Neural Network Layer Architectures

Dense (Fully Connected) Layers

The **Dense layer** is the fundamental building block studied throughout this chapter. Every neuron in a dense layer receives input from all neurons in the previous layer:

$$a_j^{[l]} = g\left(\vec{w}_j^{[l]} \cdot \vec{a}^{[l-1]} + b_j^{[l]}\right). \tag{5.24}$$

Limitation.. The number of parameters grows proportionally to the product of input and output sizes, which can be prohibitive for high-dimensional data (e.g., raw images) and risks overfitting.

Convolutional Layers

In a **Convolutional Layer**, each neuron connects only to a small *local receptive field* of the previous layer rather than the entire input.

Advantages..

1. **Parameter efficiency**: Shared filters drastically reduce the number of trainable parameters.
2. **Translation equivariance**: The same filter detects features (edges, textures) wherever they appear in the input.
3. **Reduced overfitting**: Fewer parameters mean less capacity to memorise noise.

1D Convolutional Layers for Time-Series

Convolutional layers are not limited to 2D images; they are highly effective for 1D sequential data such as EKG signals.

Example 5.4.2 (EKG Classification)**.** An EKG signal consists of, say, 100 voltage readings over time. Rather than connecting every neuron to all 100 readings (dense), a 1D convolutional neuron might look at a window of 20 consecutive readings. Successive neurons look at shifted windows. Multiple stacked layers detect increasingly abstract cardiac patterns, before a final sigmoid or softmax layer produces a diagnosis.

5.5 Backpropagation

5.5.1 The Intuition Behind Derivatives

Backpropagation is the algorithm that computes the partial derivatives of the cost J with respect to every parameter in the network. These derivatives are then used by the optimiser (gradient descent or Adam) to update the parameters.

Definition 5.5.1 (Derivative (informal))**.** The **derivative** $\frac{\partial J}{\partial w}$ is the rate of change of J with respect to w. If increasing w by a small amount ε causes J to change by approximately $k\varepsilon$, then $\frac{\partial J}{\partial w} \approx k$.

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

w	$J = w^2$	$w + \varepsilon$	ΔJ	$\frac{\partial J}{\partial w} = 2w$
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 two things:

1. **Direction**: positive ⇒ decrease w to reduce J; negative ⇒ increase w.
2. **Magnitude**: large derivative ⇒ take a large step; small derivative ⇒ take a small step.

5.5.2 Computation Graphs

Definition 5.5.2 (Computation Graph)**.** A **computation graph** is a directed acyclic graph (DAG) in which each node represents an elementary mathematical operation and each edge carries a value (tensor) between operations. Forward propagation traverses the graph left-to-right; backpropagation traverses it right-to-left.

Forward Pass: A Worked Example

Consider a single-unit network with linear activation: $a = wx + b$, and squared-error cost $J = \frac{1}{2}(a - y)^2$.

Given values:. $x = -2$, $y = 2$, $w = 2$, $b = 8$.

$$c = w \cdot x = 2 \cdot (-2) = -4, \tag{5.25}$$

$$a = c + b = -4 + 8 = 4, \tag{5.26}$$

$$d = a - y = 4 - 2 = 2, \tag{5.27}$$

$$J = \tfrac{1}{2}d^2 = \tfrac{1}{2}(4) = 2. \tag{5.28}$$

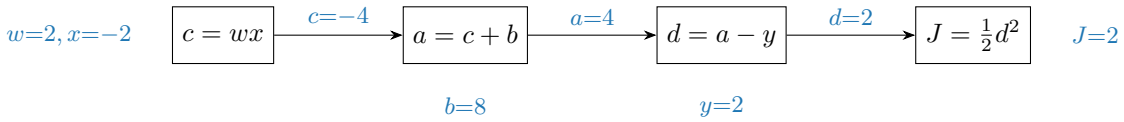


Figure 5.5. Computation graph for the forward pass. Values flow left-to-right.

Backward Pass: Applying the Chain Rule

Backpropagation traverses the graph in reverse, applying the **Chain Rule** of calculus at each node. Recall the Chain Rule:

$$\frac{\partial J}{\partial u} = \frac{\partial J}{\partial v} \cdot \frac{\partial v}{\partial u}, \tag{5.29}$$

where u feeds into v which feeds into J.

Step-by-step backward pass..

$$\frac{\partial J}{\partial d} = d = 2. \tag{5.30}$$

$$\frac{\partial J}{\partial a} = \frac{\partial J}{\partial d} \cdot \frac{\partial d}{\partial a} = 2 \cdot 1 = 2. \tag{5.31}$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial a} \cdot \frac{\partial a}{\partial b} = 2 \cdot 1 = 2. \tag{5.32}$$

$$\frac{\partial J}{\partial c} = \frac{\partial J}{\partial a} \cdot \frac{\partial a}{\partial c} = 2 \cdot 1 = 2. \tag{5.33}$$

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial c} \cdot \frac{\partial c}{\partial w} = 2 \cdot x = 2 \cdot (-2) = -4. \tag{5.34}$$

Verification (nudge test).. With $w = 2.001$: $a = 2.001 \cdot (-2) + 8 = 3.998$, $d = 1.998$, $J = \frac{1}{2}(1.998)^2 \approx 1.9960$. The change is $\Delta J \approx -0.004 = -4 \times 0.001 = \dfrac{\partial J}{\partial w} \cdot \varepsilon$. ✓

5.5.3 Backpropagation in a Deeper Network

Architecture and Forward Pass

Parameters and inputs.. $x = 1$, $y = 5$; Layer 1: $w^{[1]} = 2$, $b^{[1]} = 0$; Layer 2: $w^{[2]} = 3$, $b^{[2]} = 1$. Activation: ReLU.

$$z^{[1]} = w^{[1]}x + b^{[1]} = 2, \qquad a^{[1]} = \max(0, 2) = 2, \tag{5.35}$$

$$z^{[2]} = w^{[2]}a^{[1]} + b^{[2]} = 7, \qquad a^{[2]} = \max(0, 7) = 7, \tag{5.36}$$

$$J = \tfrac{1}{2}(a^{[2]} - y)^2 = \tfrac{1}{2}(7 - 5)^2 = 2. \tag{5.37}$$

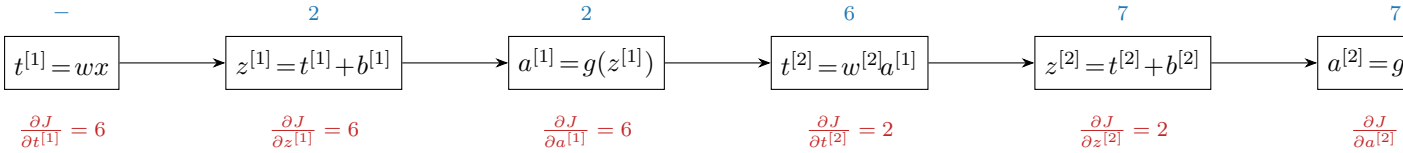


Figure 5.6. Computation graph with forward values (blue, top) and backward gradients (red, bottom). The gradient flows right-to-left via the Chain Rule.

Full Computation Graph

Key Derivative: $\dfrac{\partial J}{\partial w^{[1]}} = 6$

Derivation..

$$\frac{\partial J}{\partial a^{[2]}} = a^{[2]} - y = 7 - 5 = 2. \tag{5.38}$$

$$\frac{\partial J}{\partial z^{[2]}} = \frac{\partial J}{\partial a^{[2]}} \cdot \underbrace{\frac{\partial a^{[2]}}{\partial z^{[2]}}}_{\text{ReLU deriv.}=1} = 2 \cdot 1 = 2. \tag{5.39}$$

$$\frac{\partial J}{\partial a^{[1]}} = \frac{\partial J}{\partial z^{[2]}} \cdot w^{[2]} = 2 \cdot 3 = 6. \tag{5.40}$$

$$\frac{\partial J}{\partial w^{[1]}} = \frac{\partial J}{\partial a^{[1]}} \cdot x = 6 \cdot 1 = 6. \tag{5.41}$$

Verification.. With $w^{[1]} = 2.001$: $a^{[1]} = 2.001$, $z^{[2]} = 7.003$, $a^{[2]} = 7.003$, $J_{new} = \frac{1}{2}(2.003)^2 \approx 2.006$. Change: $\Delta J \approx 0.006 = 6 \times 0.001$. ✓

5.5.4 Computational Efficiency of Backpropagation

Naive Numerical Differentiation vs. Backpropagation

Method	Complexity	Description
Naive (numerical)	$\mathcal{O}(N \times P)$	Re-run the entire forward pass for each parameter
Backpropagation	$\mathcal{O}(N + P)$	Single backward pass reusing partial derivatives

where N is the number of nodes in the computation graph and P is the number of parameters.

For a network with one million parameters, the naive approach requires one million forward passes per gradient computation—physically intractable for any real training loop. Backpropagation collapses this to a single backward pass of cost comparable to one forward pass.

Remark 5.5.1 (Why backpropagation works so efficiently)**.** Each node in the backward pass computes a *local* gradient (derivative of its output with respect to its inputs) and multiplies it by the incoming gradient from the right. This reuse of previously computed quantities is what drives the $\mathcal{O}(N + P)$ complexity.

5.5.5 Automatic Differentiation (Autodiff)

Modern deep learning frameworks—TensorFlow and PyTorch—implement **automatic differentiation**: they construct the computation graph dynamically during the forward pass and then traverse it in reverse during the backward pass, computing all gradients without any user-supplied derivative formulas.

This has transformed the machine learning landscape:

- **Earlier era**: researchers derived gradient formulas by hand, coded them in C, and debugged numerical errors.
- **Modern era**: a single call to `model.fit()` invokes autodiff internally, computing exact gradients for arbitrarily complex architectures.

Understanding the underlying mathematics (the Chain Rule, the computation graph, the structure of partial derivatives) remains essential for *debugging*, since autodiff does not tell you whether a gradient is correct—only whether it could be computed. A solid conceptual foundation lets you recognise symptoms such as vanishing gradients, exploding gradients, and incorrect loss formulations.

5.5.6 Verifying Derivatives with SymPy

For small networks or during algorithm development, the Python library `sympy` can verify derivatives analytically:

```
import sympy

# Define symbolic variables

w = sympy.Symbol('w')

# Define the cost function

J = w**2

# Compute the derivative symbolically

dJ_dw = sympy.diff(J, w)    # Returns: 2*w

# Evaluate at a specific point

value_at_3 = dJ_dw.subs(w, 3)   # Returns: 6
value_at_2 = dJ_dw.subs(w, 2)   # Returns: 4
```

Listing 5.12. Using SymPy to compute and verify derivatives.

Note. SymPy computes *exact* symbolic derivatives. This is invaluable for verifying that a custom loss function or novel activation function has the gradient you expect before embedding it in a large training loop.

5.5.7 Notation Summary

Symbol	Name	When used
$\frac{d}{dw}$	Ordinary derivative	J depends on a single variable w
$\frac{\partial}{\partial w}$	Partial derivative	J depends on multiple variables $w_1, \ldots, b$

In machine learning we almost exclusively use partial derivatives, since every cost function depends on thousands of weights and biases simultaneously.

5.5.8 Chapter Summary

Concept	Key Takeaway
3-step training	Specify → Compile → Fit
Activation choice	ReLU for hidden; task-dependent for output
Linear collapse	All-linear networks ≡ 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
Backprop	$\mathcal{O}(N + P)$ vs. $\mathcal{O}(N \times P)$ naive
Autodiff	TF/PyTorch automate all gradient computation

CHAPTER 6

Advice for Applying Machine Learning

Chapter Overview. Having a working model is only the beginning. This chapter provides a systematic toolkit for *improving* models: the bias–variance trade-off, cross-validation for model selection, regularisation parameter tuning, learning curves as diagnostic tools, the iterative ML development cycle, and practical guidance on error analysis and data strategies.

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, and methods for handling skewed datasets.

6.1 Advice for Applying Machine Learning

6.1.1 The Problem: Unacceptable Prediction Errors

Suppose you have implemented **Regularized Linear Regression** to predict housing prices using the cost function:

$$J(\mathbf{w}, b) = \frac{1}{2m} \sum_{i=1}^{m} \left(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)} \right)^2 - \frac{\lambda}{2m} \sum_{j=1}^{n} w_j^2. \tag{6.1}$$

After training, if the model produces unacceptably large errors, you face a critical decision: *what should you try next?* Choosing the wrong path can result in months of wasted effort. A systematic approach is therefore essential.

6.1.2 Six Common Improvement Strategies

There are six commonly considered paths when a learning algorithm underperforms. Table 6.1 summarises them.

Table 6.1. Common improvement strategies for a poorly performing model.

Strategy	Description
Data Collection	Gather more training examples (e.g. more housing-price records).
Feature Reduction	Use a smaller feature set to prevent noise from irrelevant inputs.
Feature Expansion	Introduce additional informative features (e.g. proximity to schools).
Polynomial Complexity	Add polynomial terms x_1^2, x_2^2, x_1x_2 for non-linear patterns.
Decrease λ	Allow the model to fit training data more closely by reducing regularisation.
Increase λ	Simplify the model and reduce overfitting by strengthening regularisation.

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

6.1.3 Machine Learning Diagnostics

Definition 6.1.1 (Diagnostic)**.** A **diagnostic** is a formal test designed to gain insight into what is or is not working within a learning algorithm, providing evidence-based guidance for improvement.

The value of diagnostics lies in two properties:

- **Efficiency.** A diagnostic can reveal, for example, that collecting more data will *not* improve performance — saving potentially months of effort.
- **Strategic investment.** While diagnostics require upfront implementation time, they consistently pay off by replacing trial-and-error with targeted action.

6.1.4 Evaluating Model Performance

The Goal: Generalisation

A model can achieve near-zero error on training data — especially when using high-order polynomials — yet fail significantly on new examples. This phenomenon is called **overfitting**. In low-dimensional settings we can visualise it directly (plotting the "wiggly" curve), but for real-world problems with many features (e.g. house size, number of bedrooms, number of floors, and age) we need a mathematical, systematic evaluation method.

The Train / Test Split

The available dataset is divided into two non-overlapping subsets:

1. **Training set** ($\approx$ 70%–80% of data). Used to fit the model parameters $(\mathbf{w}, b)$. Contains m_{train} examples.
2. **Test set** ($\approx$ 20%–30% of data). Used to evaluate performance on unseen data. Contains m_{test} examples.

Procedure for Linear Regression

Step A — Fit parameters.. Minimise the regularised cost function on the training set:

$$J(\mathbf{w}, b) = \frac{1}{2m_{\text{train}}} \sum_{i=1}^{m_{\text{train}}} \left(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}\right)^2 - \frac{\lambda}{2m_{\text{train}}} \sum_{j=1}^{n} w_j^2. \tag{6.2}$$

Step B — Compute errors without regularisation.. After training, compute the raw predictive errors on both sets by *excluding* the regularisation term:

$$J_{\text{test}}(\mathbf{w}, b) = \frac{1}{2m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} \left(f_{\mathbf{w},b}(\mathbf{x} * \text{test}^{(i)}) - y * \text{test}^{(i)}\right)^2, \tag{6.3}$$

$$J_{\text{train}}(\mathbf{w}, b) = \frac{1}{2m_{\text{train}}} \sum_{i=1}^{m_{\text{train}}} \left(f_{\mathbf{w},b}(\mathbf{x} * \text{train}^{(i)}) - y * \text{train}^{(i)}\right)^2. \tag{6.4}$$

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

Procedure for Classification

Fitting parameters.. Parameters are found by minimising the regularised logistic regression cost on the training data.

Evaluation metrics.. Two methods are commonly used:

1. **Logistic loss.** Compute J_{test} and J_{train} using the standard logistic loss (without the regularisation term), analogously to the regression procedure.
2. **0/1 Misclassification error.** The most interpretable metric: the fraction of examples that are misclassified.

$$\hat{y} = \begin{cases} 1 & \text{if } f_{\mathbf{w},b}(\mathbf{x}) \geq 0.5, \\ 0 & \text{if } f_{\mathbf{w},b}(\mathbf{x}) < 0.5. \end{cases} \tag{6.5}$$

   J_{test} is then the fraction of the test set where $\hat{y}^{(i)} \neq y^{(i)}$, and J_{train} is the corresponding fraction for the training set.

6.1.5 Model Selection: The Three-Way Data Split

The Limitation of Training Error

When parameters $\mathbf{w}$ and b are fitted to a training set, J_{train} is an *optimistic* estimate of real-world performance. A model may achieve near-zero training error by memorising the data while failing entirely on new examples. Hence J_{train} is almost always lower than the true **generalisation error**.

The Flaw in Using Only a Test Set for Model Selection

A common but flawed procedure is to train multiple model variants (e.g. polynomials of degree $d = 1, 2, \ldots, 10$), evaluate each on the test set, and pick the d that minimises J_{test}. The problem is that once d is chosen *using* the test set, J_{test} is no longer an unbiased estimate of generalisation error — it has been "used up" in the decision.

The Refined Approach: Three-Way Split

To remove selection bias, the data is split into three disjoint subsets:

Table 6.2. Standard three-way data split.

Subset	Approx. Size	Purpose
Training set	60%	Fit parameters $\mathbf{w}$ and b.
Cross-validation (CV) / Dev set	20%	Select the model (choose d, λ, or NN architecture).
Test set	20%	Provide an unbiased final estimate of generalisation error.

The three corresponding error metrics (MSE, without regularisation) are:

$$J_{\text{train}}(\mathbf{w}, b) = \frac{1}{2m_{\text{train}}} \sum_{i=1}^{m_{\text{train}}} \left(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}\right)^2, \tag{6.6}$$

$$J_{\text{cv}}(\mathbf{w}, b) = \frac{1}{2m_{\text{cv}}} \sum_{i=1}^{m_{\text{cv}}} \left(f_{\mathbf{w},b}(\mathbf{x} * \text{cv}^{(i)}) - y * \text{cv}^{(i)}\right)^2, \tag{6.7}$$

$$J_{\text{test}}(\mathbf{w}, b) = \frac{1}{2m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} \left(f_{\mathbf{w},b}(\mathbf{x} * \text{test}^{(i)}) - y * \text{test}^{(i)}\right)^2. \tag{6.8}$$

Correct Model-Selection Procedure

1. **Train.** Fit $\mathbf{w}$ and b for each candidate model using the training set.

2. **Select.** Evaluate all models on the CV set. Choose the one with the lowest J_{cv}.
3. **Estimate.** Compute J_{test} for the single chosen model to obtain a fair estimate of future performance.

Example 6.1.1 (Neural network architecture selection)**.** When choosing between networks with different numbers of layers or hidden units, train each architecture on the training set, compute J_{cv} for each (often as misclassification rate), pick the architecture with the lowest J_{cv}, and report final performance with J_{test}.

6.2 Bias and Variance

Developing a machine learning system is iterative. When a model performs poorly, **bias** and **variance** analysis provides a systematic roadmap for deciding how to improve it.

6.2.1 Defining Bias and Variance

High Bias (Underfitting)

Definition 6.2.1 (High Bias)**.** A model suffers from **high bias** when it is too simple to capture the underlying patterns of the data.

Example: Fitting a straight line ($d = 1$) through non-linear data.
Diagnostic signature: Both J_{train} and J_{cv} are **high**, and $J_{\text{cv}} \approx J_{\text{train}}$.

High Variance (Overfitting)

Definition 6.2.2 (High Variance)**.** A model suffers from **high variance** when it is too complex and fits noise in the training set rather than the true underlying pattern.

Example: A high-degree polynomial ($d = 4$ or higher) that passes through every training point.
Diagnostic signature: J_{train} is **very low**, but $J_{\text{cv}} \gg J_{\text{train}}$.

"Just Right" (Good Generalisation)

A well-tuned model captures the pattern without over-complicating it. Both J_{train} and J_{cv} are low and close to each other.

6.2.2 Systematic Diagnosis Using Error Metrics

Table 6.3 summarises how to interpret the combination of J_{train} and J_{cv}:

Table 6.3. Bias–variance diagnostic based on error metrics.

Condition	J_{train}	J_{cv}	**Relationship**
High Bias	High	High	$J_{\text{train}} \approx J_{\text{cv}}$
High Variance	Low	High	$J_{\text{cv}} \gg J_{\text{train}}$
Just Right	Low	Low	$J_{\text{cv}} \approx J_{\text{train}}$
High Bias & High Variance	High	Much higher	$J_{\text{cv}} \gg J_{\text{train}}$, and J_{train} is high

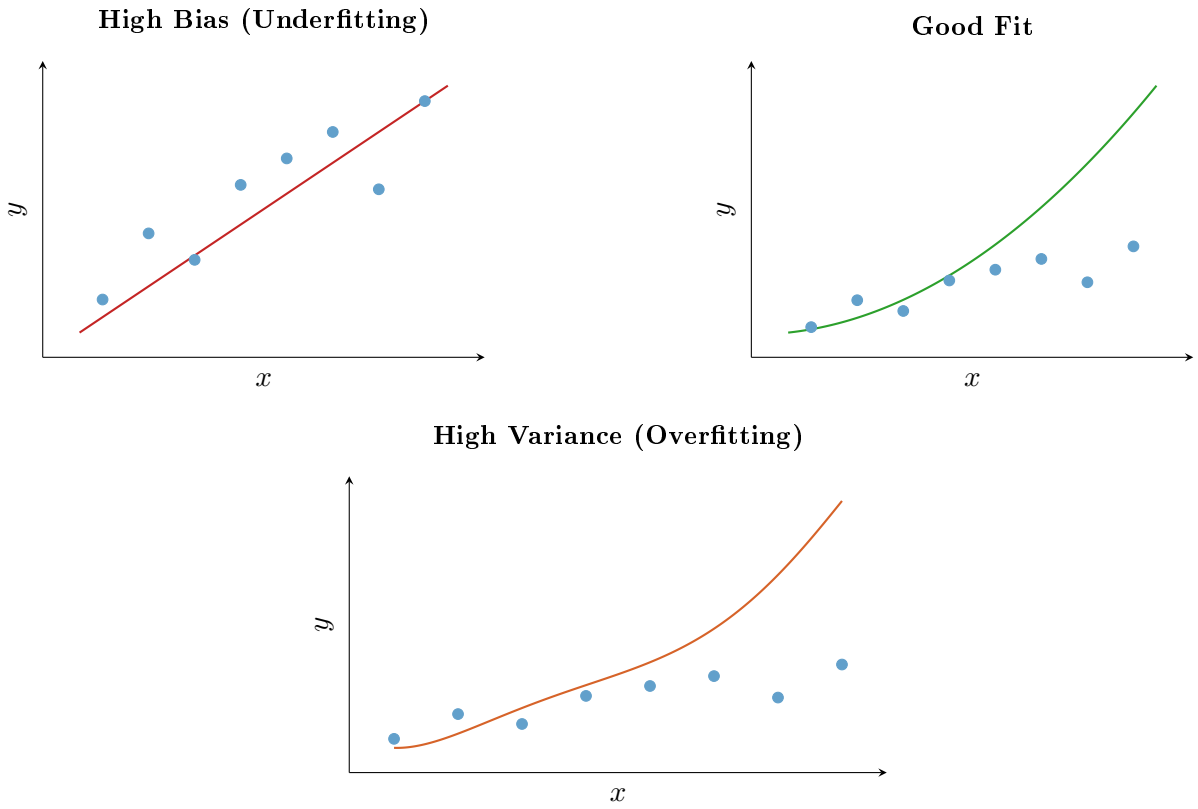


Figure 6.1. The three regimes of model fit. **Top left:** a linear model (too simple) systematically misses the curved trend—high bias. **Top right:** a quadratic model captures the underlying pattern well—good generalisation. **Bottom:** a high-degree polynomial passes through every training point but will perform poorly on new data—high variance.

6.2.3 Error vs. Model Complexity

Figure 6.2 visualises the typical behaviour of J_{train} and J_{cv} as the degree d of a polynomial model increases.

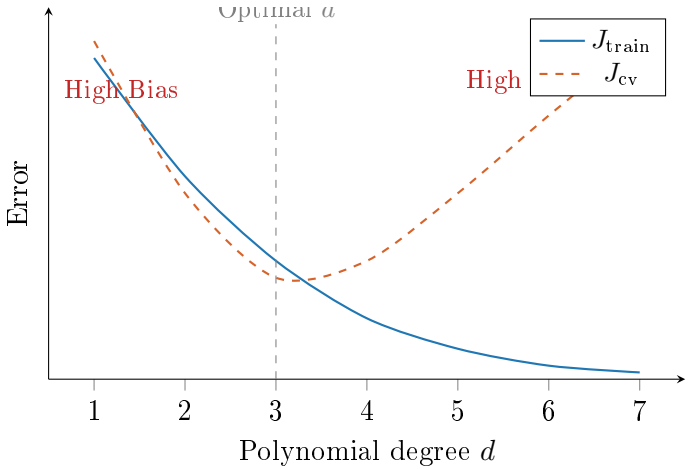


Figure 6.2. Training error (J_{train}) decreases monotonically with model complexity, while cross-validation error (J_{cv}) follows a U-shaped curve. The minimum of J_{cv} identifies the optimal model complexity.

Key observations:

- $J_{\textbf{train}}$**:** Consistently *decreases* as d increases — a more complex model can always fit the training data more closely.
- $J_{\textbf{cv}}$**:** Follows a *U-shaped* curve. The left side (low d) corresponds to high bias; the right side (high d) to high variance; the valley is the "sweet spot."

6.2.4 Regularisation and the Bias–Variance Trade-off

The regularisation parameter λ in equation (6.1) controls the trade-off between fitting the training data and keeping parameter values small.

Impact of λ on Performance

- **Large** λ (e.g. $\lambda = 10{,}000$): The algorithm is heavily penalised for large weights, forcing $\mathbf{w} \approx \mathbf{0}$. The model degenerates to a near-constant $f(\mathbf{x}) \approx b$. *Result:* **High Bias** — both J_{train} and J_{cv} are high.
- **Small** λ (e.g. $\lambda = 0$): Almost no penalty for complex parameters. The model captures noise (e.g. a high-order polynomial wiggles through every training point). *Result:* **High Variance** — J_{train} is low but J_{cv} ggJ_{train}.
- **Intermediate** λ: Balances the two extremes. *Result:* "**Just Right**" — both errors are relatively low.

Choosing λ via Cross-Validation

1. **Define candidates.** Build a list of λ values, e.g. $\{0,\ 0.01,\ 0.02,\ 0.04,\ 0.08,\ \ldots,\ 10\}$ (each step doubles the previous).
2. **Train.** For each $\lambda^{(i)}$, minimise $J(\mathbf{w}, b)$ on the training set to obtain $\mathbf{w}^{(i)}, b^{(i)}$.
3. **Evaluate.** Compute J_{cv} for each parameter set (*without* the regularisation term).
4. **Select.** Choose the λ that yields the lowest J_{cv}.
5. **Estimate generalisation.** Report the performance of the chosen model using J_{test}.

Figure 6.3 illustrates the corresponding error curves.

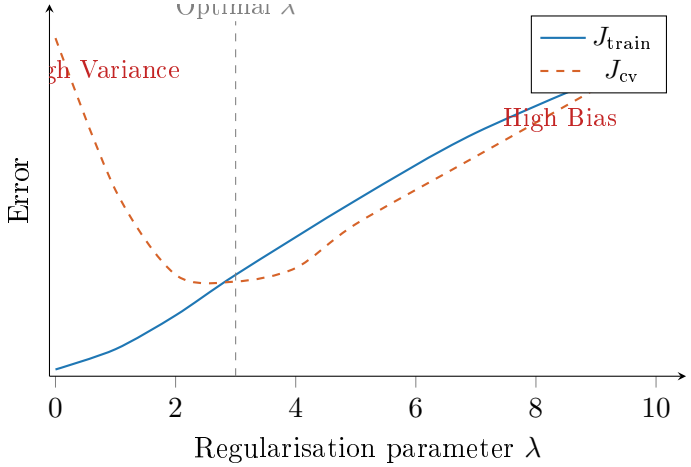


Figure 6.3. As λ increases, J_{train} rises (harder to fit data) while J_{cv} forms a U-shape. The valley of J_{cv} locates the optimal λ.

6.2.5 Establishing a Baseline for Diagnosis

Relying solely on the *absolute* values of J_{train} can be misleading. Many real-world tasks have an irreducible noise floor — even the best possible system cannot achieve 0% error.

Defining Baseline Performance

The baseline is the lowest error you can *reasonably expect* to achieve. Common sources include:

- **Human-level performance** — highly effective for unstructured data (audio, images, text).
- **Competing algorithms** — published results from existing systems.
- **Experience-based estimates** — informed guesses from similar past tasks.

Diagnosis via "The Gaps"

Definition 6.2.3 (Bias Gap and Variance Gap)**.**

$$\text{Bias Gap} = J_{\text{train}} - \text{Baseline Performance}, \tag{6.9}$$

$$\text{Variance Gap} = J_{\text{cv}} - J_{\text{train}}. \tag{6.10}$$

A **large Bias Gap** indicates High Bias (underfitting); a **large Variance Gap** indicates High Variance (overfitting).

Example 6.2.1 (Speech recognition)**.** Suppose a speech model gives $J_{\text{train}} = 10.8\%$, $J_{\text{cv}} = 14.8\%$, and human-level performance is 10.6%.

- Bias Gap $= 10.8 - 10.6 = 0.2\%$ — *small*: the model fits training data nearly as well as humans.
- Variance Gap $= 14.8 - 10.8 = 4.0\%$ — *large*: the model struggles to generalise.
- **Conclusion:** High Variance problem.

Table 6.4 illustrates several scenarios with a 10.6% baseline.

Table 6.4. Scenario analysis using the gap framework (baseline = 10.6%).

Case	J_{train}	J_{cv}	**Bias Gap**	**Variance Gap**	**Verdict**
A	10.8%	14.8%	0.2% (Low)	4.0% (High)	**High Variance**
B	15.0%	15.5%	4.4% (High)	0.5% (Low)	**High Bias**
C	15.0%	19.7%	4.4% (High)	4.7% (High)	**High Bias & Variance**

6.2.6 Learning Curves

Definition 6.2.4 (Learning Curve)**.** A **learning curve** plots J_{train} and J_{cv} as a function of the number of training examples m_{train}.

General Behaviour

In a typical, healthy learning scenario:

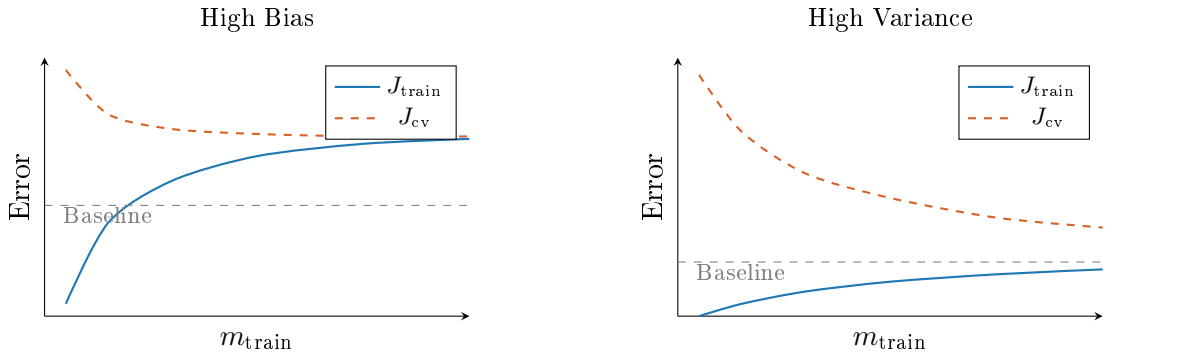
- J_{train}**:** Generally *increases* with m_{train}. With very few examples a model can fit them perfectly; as more data is added, the average error rises.
- J_{cv}**:** Generally *decreases* with m_{train}, as the model learns better-generalising parameters.
- The gap $J_{\text{cv}} > J_{\text{train}}$ is expected because the model is optimised on the training set.

High Bias Learning Curve

Both J_{train} and J_{cv} *plateau* quickly at a high error level, and the gap between them is small. Crucially, **adding more data will not help**: a model that is too simple (e.g. a straight line fitted to curved data) cannot capture the pattern regardless of how many examples it sees.

High Variance Learning Curve

There is a *large gap* between the low J_{train} and the high J_{cv}. Adding more data **is likely to help**: as m_{train} increases, the model is forced to learn a more general pattern, J_{train} rises slightly, J_{cv} drops, and the gap closes.



(a) Both errors plateau at a high level; more data does not help.

(b) Large gap between errors; more data closes the gap and helps.

Figure 6.4. Learning curves for high-bias (left) and high-variance (right) models.

Remark 6.2.1. Although theoretically powerful, plotting learning curves is *computationally expensive* as it requires retraining the model multiple times on progressively larger subsets. In practice, use learning curves primarily as a **mental model** to guide intuition.

6.2.7 Debugging Strategies: Applying the Diagnosis

Once the root cause is identified, the following targeted interventions apply.

Table 6.5. Recommended fixes based on bias–variance diagnosis.

Diagnosis	Recommended Strategy
High Variance	Get more training examples
	Try a smaller feature set
	Increase regularisation parameter λ
High Bias	Add more / better features
	Add polynomial features (x^2, x_1x_2, etc.)
	Decrease regularisation parameter λ

6.2.8 Neural Networks and Bias–Variance

Neural networks have altered the traditional bias–variance *trade-off* because a sufficiently large network can fit almost any function, making it effectively a **low-bias machine**. The iterative development workflow for neural networks is:

1. **Check J_{train}:** If high relative to baseline $\Rightarrow$ **High Bias**. Action: use a *bigger network* (more layers or hidden units). Repeat.
2. **Check J_{cv}:** If $J_{\text{cv}} \gg J_{\text{train}} \Rightarrow$ **High Variance**. Action: *get more data*. Repeat.
3. **Done:** Once both J_{train} and J_{cv} are acceptably low.

The key insight is that a larger, *well-regularised* network typically performs as well as or better than a smaller one. Regularisation is applied per layer via an L2 penalty:

$$J(\mathbf{W},\mathbf{B}) = \frac{1}{m}\sum_{i=1}^{m} L\big(f(\mathbf{x}^{(i)}), y^{(i)}\big) - \frac{\lambda}{2m}\sum_{\ell=1}^{L}\sum_{j}\sum_{k}\big(w_{j,k}^{[\ell]}\big)^2. \tag{6.11}$$

A practical TensorFlow/Keras implementation:

```
from tensorflow.keras.layers import Dense
from tensorflow.keras.regularizers import L2
from tensorflow.keras import Sequential

layer_1 = Dense(units=25, activation="relu", kernel_regularizer=L2(0.01))
layer_2 = Dense(units=15, activation="relu", kernel_regularizer=L2(0.01))
layer_3 = Dense(units=1,  activation="sigmoid", kernel_regularizer=L2(0.01)
    )

model = Sequential([layer_1, layer_2, layer_3])
```

Listing 6.1. Regularised neural network in Keras.

The main trade-off of using a larger, regularised network is **computational cost** (slower training and inference) rather than predictive accuracy.

6.3 Machine Learning Development Process

6.3.1 The Iterative Development Loop

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

1. **Choose architecture.** Select the model family (e.g. logistic regression, neural network), the data and features, and the hyperparameters (e.g. λ).
2. **Train model.** Implement and train the chosen architecture. Initial models rarely meet performance goals immediately.
3. **Run diagnostics.** Perform bias–variance analysis and error analysis. Use the findings to loop back and refine the architecture.

Figure 6.5 depicts this cycle along with a complete ML project lifecycle.

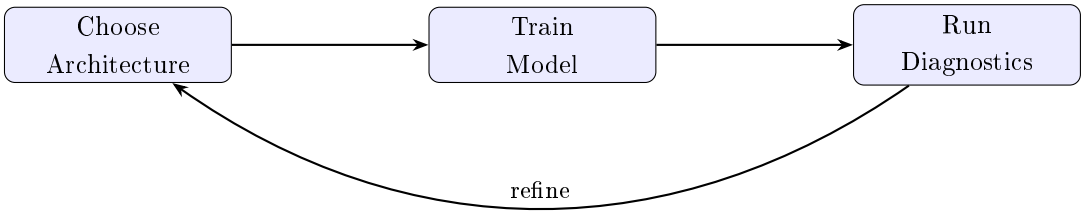


Figure 6.5. The iterative loop of ML model development.

6.3.2 The Four Stages of an ML Project Lifecycle

A complete ML project spans from conception to long-term maintenance:

Stage 1. Project Scoping. Define the problem and goals. What specific task should the AI solve?

Stage 2. Data Collection. Gather inputs x and labels y (e.g. audio clips and their text transcripts for speech recognition).

Stage 3. Model Training. Build, train, and refine using error analysis. Findings often dictate a return to data collection for targeted acquisition or augmentation.

Stage 4. Deployment in Production. Integrate the model via an inference server, monitor for data drift, and update as needed.

6.3.3 Deployment Architecture and MLOps

Once trained, a model is typically deployed as an **inference server** that responds to API calls:

- A client (e.g. mobile app) sends input x.

- The server runs the model and returns prediction $\hat{y}$.

MLOps (Machine Learning Operations) addresses the engineering challenges of this deployment:

- **Scaling:** Handling high request volumes efficiently.
- **Reliability:** Ensuring consistent, fast predictions.
- **Logging:** Recording inputs and outputs for future analysis (subject to privacy/consent constraints).
- **Monitoring:** Detecting *data drift* — degradation caused by real-world data shifting away from the training distribution (e.g. new slang terms not in the original vocabulary).
- **Model updates:** Systematically replacing older models with retrained versions.

6.3.4 Case Study: Email Spam Classification

Spam classification is a classic supervised learning problem for text data.

Problem Definition

- **Input $\mathbf{x}$**: Features derived from email content and metadata.
- **Output** y: $1 =$ Spam, $0 =$ Ham (non-spam).

Feature Engineering via Vocabulary Vectors

1. Select a dictionary of the top 10,000 most frequent words.
2. Construct a feature vector $\mathbf{x} \in \mathbb{R}^{10{,}000}$.
3. Assign $x_j = 1$ if word j appears in the email (binary), or $x_j = n$ where n is the word count.

Improvement Strategies

- **Data collection via honeypots:** Use fake email addresses to lure spammers and collect labelled spam examples.
- **Email routing features:** Analyse email headers to identify suspicious server paths frequently used by spammers.
- **Sophisticated body features:** Apply *stemming/lemmatisation* (treating 'discount" and' discounting" as the same token) and contextual analysis.
- **Misspelling detection:** Build features to recognise deliberate character substitutions (e.g. 'w4tches",' m0rtgage").

Remark 6.3.1. Choosing the correct improvement path based on diagnostics — rather than intuition — can accelerate project development by an estimated **10**× compared to trial-and-error.

6.3.5 Data-Centric AI and Augmentation Strategies

The Shift Toward Data-Centric AI

Historically, AI research held the dataset fixed and focused on algorithmic improvements. Modern highly capable algorithms (deep neural networks, gradient-boosted trees) have shifted attention toward **data quality and quantity**: hold the algorithm fixed and engineer, improve, and expand the data.

Targeted Data Collection

Rather than gathering data indiscriminately, use error analysis to **identify weaknesses** and then target data collection *only* in those areas.

Example 6.3.1. If a spam filter performs well overall but struggles with pharmaceutical spam, direct labellers to find and annotate specifically "pharma spam" from an unlabelled pool — not random emails.

Data Augmentation

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 (only when the label remains valid), contrast changes, and grid-based warping/distortion.
- **Speech recognition:** Adding background noise (crowd noise, car engines), or applying filters to simulate poor phone connections.

The Representative Noise Rule: Augmentation distortions must be *representative of real test-set noise*. Purely random pixel noise, for instance, is rarely seen in practice and provides little benefit.

Artificial Data Synthesis

Definition 6.3.2 (Data Synthesis)**. Data synthesis** generates entirely new training examples from scratch rather than transforming existing ones.

Example 6.3.2 (Photo OCR)**.** Instead of photographing real text, programmatically render characters in diverse fonts, colours, and backgrounds to create a large synthetic dataset for an OCR system.

Transfer Learning

When targeted collection, augmentation, and synthesis are insufficient due to extreme data scarcity, **transfer learning** is the recommended strategy.

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, low-data task.

The standard two-step workflow is:

1. **Supervised pre-training:** Train a large network on a massive dataset (e.g. 1,000,000 labelled images).
2. **Fine-tuning:** Replace the output layer with one suited to the target task (e.g. 10 units for digits 0–9). Then either:
   - **Frozen layers (Option 1):** Train *only* the new output layer. Best when the new dataset is very small ($\approx$ 50–100 examples).
   - **Full fine-tuning (Option 2):** Retrain *all* layers using the pre-trained weights as initialisation. Best when the new dataset is moderate to large.

Transfer learning works because neural networks learn a **hierarchy of features**: early layers detect universal low-level patterns (edges, corners) that are task-agnostic, while later layers combine them into task-specific concepts. By reusing early layers the model avoids re-learning these universal building blocks.

Key constraints:

- The input type x must be *identical* for both tasks (e.g. image models for images; text models such as BERT or GPT for text).
- Pre-trained weights are widely available from research communities (e.g. ImageNet models).

6.3.6 Ethics and Fairness in Machine Learning

Core Concerns

As ML systems impact billions of users, ensuring **fairness**, detecting **algorithmic bias**, and adopting an **ethical development approach** are mandatory parts of the development lifecycle — not optional additions.

Historical Examples of Algorithmic Bias

- **Hiring discrimination:** Systems trained on historical data have penalised résumés from women's colleges or mentioning "women's" organisations.
- **Facial recognition disparities:** Commercial systems have shown error rates as low as 0.8% for lighter-skinned males versus nearly 34.7% for darker-skinned females (*Gender Shades* study).
- **Financial bias:** Automated loan systems have denied credit to specific ethnicities at disproportionate rates despite comparable financial profiles.
- **Stereotype reinforcement:** Search and recommendation systems can amplify existing social stereotypes.

Adverse Use Cases

- **Deepfakes:** Generative models creating realistic but fabricated audio/video without consent, enabling fraud and misinformation.
- **Engagement optimisation:** Social media algorithms that maximise engagement may inadvertently amplify toxic or inflammatory content.
- **Automated malice:** Bots for fake reviews, political manipulation, or financial fraud.

A Framework for Ethical ML Development

I. **Team diversity:** Assemble teams diverse in gender, ethnicity, culture, and background to increase the likelihood of identifying harms to under-represented groups.
II. **Literature and standards review:** Consult industry-specific guidelines (e.g. financial or medical regulations) and emerging technical definitions of fairness.
III. **Pre-deployment auditing:** Measure model performance across demographic subgroups. Flag cases where error rates differ significantly.
IV. **Mitigation and monitoring:** Prepare a rollback strategy and continue monitoring for real-world harms post-deployment.

If a project is financially viable but poses clear ethical harms, the professional recommendation is to **decline or terminate the project**.

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. a fraud detection dataset where 99% of cases are "not fraud").

On a skewed dataset, a naive classifier that *always* predicts the majority class can achieve very high accuracy while providing no useful predictions. For example, in a dataset where only 1% of patients have a rare disease, always predicting "negative" yields 99% accuracy — yet the classifier is completely useless. This motivates the use of **Precision** and **Recall** instead.

6.4.2 The Confusion Matrix

Performance on a binary classification problem is summarised by four counts:

Table 6.6. Confusion matrix for binary classification.

		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)

6.4.3 Precision and Recall

Definition 6.4.2 (Precision)**. Precision** is the fraction of predicted positives that are actually positive:

$$\text{Precision} = \frac{TP}{TP + FP}. \tag{6.12}$$

Definition 6.4.3 (Recall)**. Recall** (sensitivity) is the fraction of actual positives that were correctly identified:

$$\text{Recall} = \frac{TP}{TP + FN}. \tag{6.13}$$

A classifier that always predicts "negative" has Recall = 0 (it never identifies any true positive), correctly exposing its uselessness despite high accuracy.

6.4.4 The Precision–Recall Trade-off

In logistic regression, the default decision threshold is 0.5:

$$\hat{y} = \begin{cases} 1 & f_{\mathbf{w},b}(\mathbf{x}) \geq \text{threshold}, \\ 0 & f_{\mathbf{w},b}(\mathbf{x}) < \text{threshold}. \end{cases} \tag{6.14}$$

Adjusting the threshold navigates the trade-off between precision and recall:

- **Higher threshold** (e.g. 0.7–0.9, 'conservative"): Increases precision, decreases recall. Appropriate when false positives have severe consequences (e.g. recommending an unnecessary invasive procedure).
- **Lower threshold** (e.g. 0.3,' aggressive"): Increases recall, decreases precision. Appropriate when false negatives are costly (e.g. missing a fatal, highly contagious disease) and treatment is safe/cheap.

The Precision–Recall Curve

Sweeping the threshold from near 1 to near 0 traces a **Precision–Recall curve** (Figure 6.6).

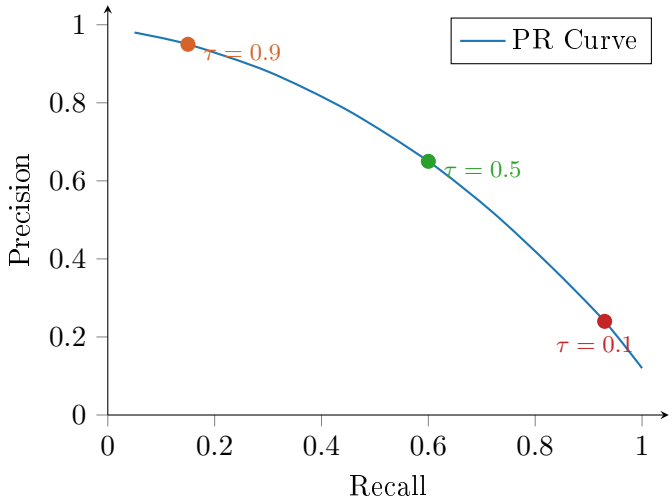


Figure 6.6. A typical Precision–Recall curve. Moving from right to left (increasing threshold τ) increases precision at the cost of reduced recall.

6.4.5 The F1 Score

When comparing multiple classifiers, evaluating two metrics simultaneously is cumbersome. A single composite score is needed.

Why Not Use the Arithmetic Mean?

The arithmetic average $\frac{P+R}{2}$ is a poor metric because it does not penalise extreme imbalance. A classifier that predicts $\hat{y} = 1$ for every example achieves Recall $= 1.0$ but near-zero Precision; the arithmetic mean remains deceptively inflated.

The F1 Score (Harmonic Mean)

Definition 6.4.4 (F1 Score)**.** The **F1 score** is the harmonic mean of Precision and Recall:

$$F_1 = \frac{1}{\frac{1}{2}\left(\frac{1}{P} + \frac{1}{R}\right)} = \frac{2PR}{P+R}. \tag{6.15}$$

The harmonic mean penalises extreme values: a high F1 score requires *both* P and R to be reasonably high. If either is near zero, the F1 score collapses towards zero regardless of the other.

Comparison Example

Table 6.7. Comparing classifiers using Arithmetic Mean vs. F1 Score.

Algorithm	Precision P	Recall R	Arith. Mean	F1 Score
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 score of 0.039 correctly identifies it as a poor classifier: it predicts "positive" for essentially everything, achieving recall = 1.0 but near-zero precision. Algorithm 1, with the most balanced performance, is correctly ranked highest by the F1 score.

6.4.6 Subgroup Analysis and Ethical Implications

Skewed datasets are also central to fairness concerns. A model may achieve high overall accuracy while performing poorly on a minority subgroup — and a high global metric masks this failure. Best practice includes:

- **Per-subgroup precision and recall:** Compute P, R, and F_1 separately for each demographic or class-relevant subgroup.
- **Disparity detection:** Flag cases where performance metrics differ significantly across subgroups and investigate root causes (e.g. under-representation of a group in the training data).
- **Rebalancing strategies:** Consider oversampling under-represented classes, adjusting decision thresholds per subgroup, or targeted data collection to address imbalances.

Chapter Summary

This chapter has developed a comprehensive framework for diagnosing and improving machine learning systems.

- **Diagnostics** replace guessing with evidence. The three-way data split (training / cross-validation / test) enables unbiased model selection and performance estimation.
- **Bias and variance** are the two primary failure modes. High bias (underfitting) calls for a more expressive model or richer features; high variance (overfitting) calls for more data, fewer features, or stronger regularisation.
- Comparing errors against a **baseline** (e.g. human-level performance) reveals whether gaps are meaningful.

- **Learning curves** clarify whether collecting more data is likely to help.
- The **iterative development loop** — architecture, training, diagnostics — is the standard workflow. Modern neural networks shift emphasis toward data quality via augmentation, synthesis, and transfer learning.
- **Skewed datasets** require Precision, Recall, and F1 score rather than raw accuracy. The precision–recall trade-off is controlled via the decision threshold.
- **Ethics and fairness** must be embedded throughout the development lifecycle, from team diversity to pre-deployment subgroup audits and post-deployment monitoring.

CHAPTER 7

Decision Trees

Chapter Overview. Decision trees and tree ensembles dominate structured/tabular data competitions. This chapter builds the theory from scratch: entropy and information gain for node splitting, the CART algorithm, pruning for regularisation, and the ensemble methods—bagging, random forests, and gradient-boosted trees (XGBoost)—that transform a single tree into a powerful predictor.

7.1 Decision Trees

7.1.1 Introduction

Decision trees are among the most powerful and interpretable machine learning algorithms in widespread use. Unlike linear models, they can naturally capture complex, nonlinear relationships in data without requiring feature scaling or distribution assumptions. They serve as the fundamental building block for more advanced *tree ensemble* methods such as Random Forests and XGBoost, which consistently rank among the top-performing algorithms on structured data competitions and real-world applications.

Remark 7.1.1. Decision trees are particularly effective for **tabular (structured) data** — data organised into rows and columns, as in a spreadsheet or database table. For unstructured data such as images, audio, or raw text, neural network architectures are generally preferred.

7.1.2 Dataset Representation

To train a decision tree, the input must be a *labelled dataset* in which each example is described by a fixed set of features and is associated with a known target label.

Data Components

Consider a concrete classification task: determining whether an animal is a cat or not, given three observable features.

- **Input features** $\mathbf{x} = (x_1, x_2, x_3)$:
  - x_1: Ear Shape — Pointy or Floppy
  - x_2: Face Shape — Round or Not Round
  - x_3: Whiskers — Present or Absent

- **Target label** y: 1 for ‘Cat”, 0 for‘ Not Cat”.
- **Data type**: The three features above are *categorical* (discrete), meaning they take on a finite set of named values rather than a continuous numeric range.

Feature Types

Decision trees can handle both categorical and continuous features. For continuous features, the algorithm tests whether a value lies above or below a learned numeric threshold (e.g., $x_4 \leq 3.5$ kg). For categorical features with more than two values, a preprocessing technique called *one-hot encoding* (discussed in Section 7.2.5) converts each category into a binary indicator variable.

7.1.3 Anatomy of a Decision Tree

A decision tree is a hierarchical, acyclic graph that encodes a sequence of *if–then–else* rules. Figure 7.1 shows a simple example for the cat classification task.

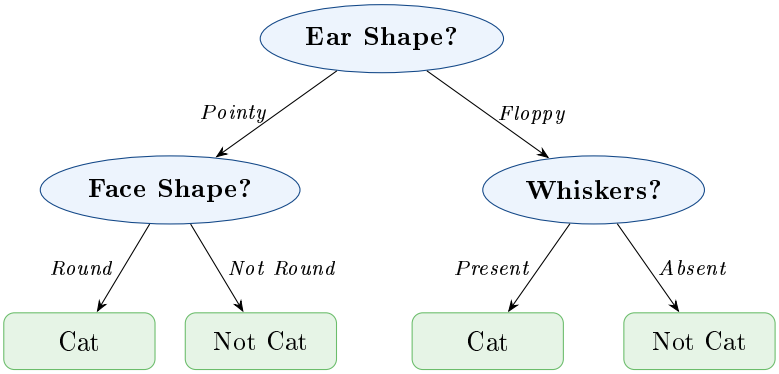


Figure 7.1. A decision tree for cat classification. Oval nodes are decision nodes; rounded rectangular nodes are leaf nodes. Each edge is labelled with the feature value that directs an example along that branch.

Node Types

Root Node

The topmost node. It receives the full training dataset and performs the very first split. There is exactly one root node.

Internal (Decision) Nodes

Intermediate nodes that test a specific feature and direct examples to one of two (or more) child branches based on the outcome.

Leaf Nodes

Terminal nodes that produce the final prediction. A leaf does not split further; instead it stores a class label (for classification) or a numeric value (for regression).

Remark 7.1.2. In computer science, trees are conventionally drawn *upside down*: the root appears at the top and the leaves at the bottom.

Tree Depth

The **depth** of a node is the number of edges on the path from the root to that node. The root node is at depth 0. The maximum depth of any leaf is called the *depth of the tree*.

Example 7.1.1. In Figure 7.1, the root node ('Ear Shape?") is at depth 0; the two intermediate nodes ('Face Shape?" and "Whiskers?") are at depth 1; and all four leaf nodes are at depth 2.

7.1.4 Inference: Classifying New Examples

Once a tree has been trained, classifying a new example is straightforward:

1. **Start at the root node.**
2. **Evaluate the feature** tested at the current node.
3. **Follow the matching branch** (e.g., if Ear Shape = "Pointy", move left).
4. **Repeat** steps 2–3 at each subsequent node.
5. **Return the prediction** stored in the reached leaf node.

Inference is therefore $O(d)$ in the depth d of the tree, which is very efficient.

7.1.5 The Learning Objective

For any given dataset, infinitely many different decision trees can be constructed by varying which features are tested and in which order. The learning algorithm must find a tree that

1. accurately predicts the *training* set, and
2. *generalises* to unseen examples — i.e., achieves low error on a held-out test or validation set.

A tree that is too deep may memorise noise in the training data (**overfitting**), while a tree that is too shallow may fail to capture the true underlying pattern (**underfitting**). The next chapter on learning algorithms describes how to balance these competing objectives.

7.2 Decision Tree Learning

7.2.1 The Recursive Splitting Algorithm

Building a decision tree proceeds in a top-down, greedy, recursive fashion: at each node, the algorithm selects the feature that produces the best partition of the local training subset, then recursively applies the same procedure to each child subset.

Step-by-Step Construction

1. **Initialise.** Place the entire training set at the root node.
2. **Feature selection.** Compute an impurity-reduction score for every available feature and choose the one that best separates the classes (detailed in Section 7.2.3).
3. **Split.** Partition the examples at the current node into subsets, one per feature value (or one per side of a numeric threshold).
4. **Recurse.** Treat each subset as a new "root" and repeat from step 2.
5. **Stop.** Apply stopping criteria (Section 7.2.4) to decide when a node should become a leaf rather than be split further.

Because step 4 calls the same routine on a strictly smaller dataset, the process is guaranteed to terminate.

7.2.2 Entropy: Measuring Impurity

Concept

At any node, the training subset may contain examples from multiple classes mixed together. The degree of mixing is called **impurity**. The algorithm seeks splits that produce child nodes as *pure* as possible — ideally containing examples from only one class.

Entropy is the standard measure of impurity in the ID3 and C4.5 decision tree algorithms. It originates in information theory, where it quantifies the amount of uncertainty (or "surprise") in a random variable.

Mathematical Definition

Let p_1 denote the fraction of examples at a node that belong to the *positive* class (e.g., "Cat"), and let $p_0 = 1 - p_1$ denote the fraction belonging to the *negative* class. The binary entropy is

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

Convention. When $p_1 = 0$ or $p_1 = 1$, the expression $0 \cdot \log_2 0$ is taken to equal 0 (consistent with the limit $\lim_{p \to 0^+} p \log_2 p = 0$).

Properties of the Entropy Function

- $H(p_1) = 0$ when $p_1 \in \{0, 1\}$: the node is perfectly pure.
- $H(p_1) = 1$ when $p_1 = 0.5$: maximum uncertainty; the two classes are equally likely.
- H is a concave, symmetric function achieving its maximum at $p_1 = 0.5$.

Figure 7.2 illustrates the entropy curve. Using base-2 logarithms normalises the peak to exactly 1, which is a convention borrowed from information theory (the unit is *bits*). Using natural

logarithms or base-10 logarithms would yield the same ordering of splits but different numerical values.

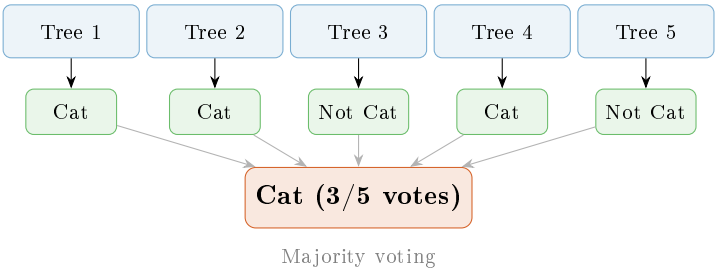


Figure 7.2. Binary entropy $H(p_1)$ as a function of the positive-class fraction p_1. The function peaks at 1 bit when the classes are balanced and vanishes at pure nodes.

Tabulated Examples

Table 7.1 provides entropy values for representative class compositions.

Table 7.1. Entropy values for different class distributions in a node of six examples.

Composition	p_1	$H(p_1)$	Impurity Level
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
2 cats, 4 dogs	0.333	0.918	Moderate
0 cats, 6 dogs	0.000	0.000	None (pure)

Alternative Impurity Measures

While entropy is the most common choice, other metrics exist:

- **Gini impurity**: $G(p_1) = p_1(1-p_1) + (1-p_1)p_1 = 2p_1(1-p_1)$. Used in the CART algorithm. Computationally cheaper (no logarithms) and behaves similarly to entropy in practice.
- **Misclassification error**: $1-\max(p_1, 1-p_1)$. Less sensitive to changes in class proportions and therefore less effective as a splitting criterion.

7.2.3 Information Gain

Motivation

Entropy alone measures the impurity at a *single* node. To evaluate the quality of a proposed split, we need to compare the impurity *before* and *after* the split. The resulting quantity is called **Information Gain (IG)**.

Formal Definition

Let n be the number of examples at the current node, n_L and n_R the numbers reaching the left and right children, and let p_1, p_1^L, p_1^R be the respective positive-class fractions. Define the weights

$$w^L = \frac{n_L}{n}, \qquad w^R = \frac{n_R}{n}, \qquad w^L + w^R = 1.$$

The Information Gain of splitting on a feature f is

$$\mathrm{IG}(f) = H(p_1) - \Big[w^L\, H(p_1^L) + w^R\, H(p_1^R)\Big]. \tag{7.2}$$

The term in brackets is the *weighted average entropy* of the two children. IG is always non-negative (by the concavity of entropy) and equals zero only when the split provides no useful information.

Why Weighted Averaging?

An impure child node containing many examples is far more costly than an impure child node containing only one or two examples. By weighting each child's entropy by the fraction of examples it receives, the formula penalises splits that route many examples into impure regions.

Worked Example

Example 7.2.1. Suppose the root node contains 10 examples: 5 cats and 5 dogs, so $H(p_1^{\text{root}}) = H(0.5) = 1$.

Split on Ear Shape:

- Left (Pointy): 5 examples, 4 cats $\Rightarrow p_1^L = 0.8$, $H = 0.72$.
- Right (Floppy): 5 examples, 1 cat $\Rightarrow p_1^R = 0.2$, $H = 0.72$.
- IG $= 1 - \left[\frac{5}{10}(0.72) + \frac{5}{10}(0.72)\right] = 1 - 0.72 = \mathbf{0.28}$.

Split on Face Shape:

- Left (Round): 7 examples, 4 cats $\Rightarrow H \approx 0.985$.

- Right (Not Round): 3 examples, 1 cat $\Rightarrow H \approx 0.918$.
- IG $= 1 - \left[\frac{7}{10}(0.985) + \frac{3}{10}(0.918)\right] \approx \mathbf{0.03}$.

Split on Whiskers:

- IG $\approx \mathbf{0.12}$.

Decision: Select *Ear Shape* as the root split, since IG(Ear) $= 0.28$ is largest.

Figure 7.3 visualises the Information Gain for each candidate feature in Example 7.2.1.

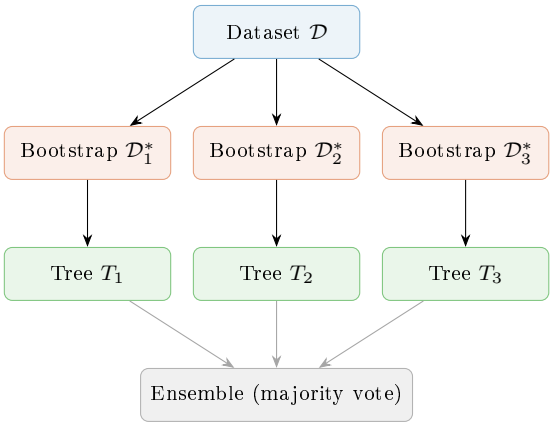


Figure 7.3. Information Gain for the three candidate features at the root node. Ear Shape achieves the highest gain and is therefore selected as the splitting feature.

7.2.4 Stopping Criteria

Without constraints, the recursive algorithm would grow a tree that perfectly classifies every training example (zero training error) at the cost of severe overfitting. Stopping criteria regulate when a node should be declared a leaf.

Common Rules

1. **Perfect purity.** Stop when all examples at a node belong to a single class ($H = 0$). This is the natural termination condition.
2. **Maximum depth.** Stop when the node depth reaches a user-specified limit $d_{\max}$. Smaller depths produce simpler, more interpretable trees with lower variance but potentially higher bias.
3. **Minimum Information Gain.** Stop if the best achievable IG at a node falls below a threshold ϵ_{IG}. Further splitting would add complexity without meaningful improvement.
4. **Minimum node size.** Stop if the number of examples at a node drops below a threshold $n_{\min}$ (e.g., $n_{\min} = 3$). This prevents splits based on too few observations.

Bias–Variance Trade-Off

The depth of the tree directly controls the bias–variance trade-off:

- A **deep** tree (small depth limit) has low training error but high variance — it fits noise (*overfitting*).
- A **shallow** tree has high bias — it cannot capture complex patterns (*underfitting*).

The optimal depth is typically selected via cross-validation on a held-out validation set.

7.2.5 One-Hot Encoding for Multi-Valued Categorical Features

Problem

The binary split framework (Section 7.2.3) assumes each feature takes at most two values. Many real-world categorical features take on $k > 2$ values (e.g., Ear Shape $\in$ {Pointy, Floppy, Oval}).

One-Hot Encoding

Definition 7.2.1 (One-Hot Encoding)**.** Given a categorical feature x with k possible values, one-hot encoding replaces x with k new binary features $b_1, \ldots, b_k$, where $b_j = 1$ if x equals the j-th category and $b_j = 0$ otherwise. Exactly one b_j is "hot" (equal to 1) for any given example.

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

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

Benefits

- **No algorithm change.** After encoding, every feature is binary, and the standard IG-based splitting procedure applies without modification.
- **Cross-model compatibility.** One-hot encoding is a universal preprocessing step. Neural networks, logistic regression, and linear models all require numerical inputs and cannot directly process string-valued features. OHE provides a principled numerical representation.

7.2.6 Regression Trees

Overview

A **regression tree** extends the decision tree framework to predict a continuous target $y \in \mathbb{R}$ (e.g., a house price, an animal's weight, or a temperature).

Prediction at Leaf Nodes

Whereas a classification leaf stores a class label, a regression leaf stores the **mean** of the target values of all training examples that reached it:

$$\hat{y} * \text{leaf} = \frac{1}{|\mathcal{S} * \text{leaf}|} \sum_{i \in \mathcal{S}_{\text{leaf}}} y_i, \tag{7.3}$$

where $\mathcal{S}_{\text{leaf}}$ is the set of training examples at the leaf.

Splitting Criterion: Variance Reduction

For regression, the goal is to minimise the *variance* of target values within each subset rather than entropy. The sample variance at a node containing examples $\{y_1, \ldots, y_n\}$ is

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} (y_i - \bar{y})^2, \qquad \bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i. \tag{7.4}$$

The **variance reduction** achieved by a split is

$$\Delta\sigma^2 = \sigma^2_{\text{root}} - \left[\frac{n_L}{n} \sigma^2_L + \frac{n_R}{n} \sigma^2_R \right]. \tag{7.5}$$

The feature (and threshold, for continuous features) that maximises $\Delta\sigma^2$ is selected.

Figure 7.4. Variance reduction in a regression tree. Splitting on "Ear Shape" separates the training examples into two groups with substantially lower within-group variance than the undivided root node. Dots represent individual training examples; dashed lines show group means.

Comparison: Classification vs. Regression Trees

Table 7.2. Key differences between classification trees and regression trees.

Aspect	**Classification Tree**	**Regression Tree**
Target y	Discrete label	Continuous number
Leaf prediction	Majority class (mode)	Mean of examples
Splitting metric	Information Gain (Entropy/Gini)	Variance Reduction
Impurity measure	Entropy / Gini	Variance

7.2.7 Full Algorithm Summary

Algorithm ?? summarises the complete decision tree construction procedure for the classification case.

Algorithm 1: Build Decision Tree ($\mathcal{D}$, depth)

Input: Training set $\mathcal{D}$, current depth; hyperparameters $d_{\max}$, $n_{\min}$, ϵ_{IG}.
Output: A (sub-)tree node.

1. **if** $\mathcal{D}$ is pure, or depth $= d_{\max}$, or $|\mathcal{D}| < n_{\min}$:
   - **return** Leaf with the majority class label.
2. **for each** feature f: compute $\text{IG}(f)$ using Equation (7.2).
3. $f^* \leftarrow \arg\max_f \text{IG}(f)$.
4. **if** $\text{IG}(f^*) < \epsilon_{\text{IG}}$: **return** Leaf with majority class.
5. Split $\mathcal{D}$ on f^* into $\mathcal{D}_L$ and $\mathcal{D}_R$.
6. **return** Node(f^*, left = BuildTree($\mathcal{D}_L$, depth+1), right = BuildTree($\mathcal{D}_R$, depth+1)).

The recursive decision tree construction algorithm.

7.3 Tree Ensembles

7.3.1 Motivation: Instability of Single Trees

A single decision tree, despite its appealing simplicity, suffers from a fundamental weakness: **high variance**. A small perturbation in the training data — even changing a single training example — can lead to a completely different tree structure with a different root split and different leaf predictions. This sensitivity makes single trees unreliable for deployment.

Remark 7.3.1. High-variance models are said to *overfit*: they fit the particular noise present in the training set rather than the underlying signal. The solution is to combine many trees, each trained on a slightly different version of the data.

7.3.2 Tree Ensembles and Majority Voting

Definition 7.3.1 (Tree Ensemble)**.** A **tree ensemble** is a collection of B decision trees $\{T_1, T_2, \ldots, T_B\}$ trained to solve the same prediction task. At inference time, each tree produces an independent prediction, and the ensemble aggregates these predictions:

$$\hat{y}_{\text{ensemble}} = \begin{cases} \text{majority class}(\hat{y}_1, \ldots, \hat{y} * B) & \text{(classification)}, \\ \dfrac{1}{B} \sum * b = 1^B \hat{y}_b & \text{(regression)}. \end{cases} \tag{7.6}$$

Figure 7.5 illustrates the majority-voting mechanism for a 5-tree ensemble.

Figure 7.5. Five decision trees vote independently on a test example. Three trees predict 'Cat" and two predict' Not Cat", so the ensemble outputs "Cat" (3 out of 5 votes).

The key insight is that independent errors tend to cancel: if each tree makes an error on a different subset of examples, the majority vote will still be correct, even when individual trees are wrong.

7.3.3 Bootstrapping and Sampling with Replacement

For an ensemble to be effective, its constituent trees must be *diverse* — they should disagree on which training patterns matter most. Diversity is engineered through **bootstrapping**, i.e., sampling with replacement.

The Bootstrapping Procedure

Given a training set $\mathcal{D}$ of size m:

1. **Draw** one example uniformly at random from $\mathcal{D}$.
2. **Record** the example and **return** it to $\mathcal{D}$ (replacement).
3. **Repeat** m times to obtain a bootstrap sample $\mathcal{D}^*$ of size m.

Because examples are returned after each draw, $\mathcal{D}^*$ will typically contain some examples multiple times and omit others entirely. On average, each bootstrap sample includes approximately 63.2% of the unique training examples (the remainder are the so-called *out-of-bag* examples, which can be used for internal validation).

Why Replacement Matters

Without replacement, drawing m items from a set of m items always produces the original set exactly — there is no diversity. With replacement, the resulting datasets are "similar but different", inducing trees with varied structure and different error profiles.

7.3.4 Bagged Decision Trees (Bootstrap Aggregating)

Bagging (*Bootstrap Aggregating*) builds an ensemble by training an independent decision tree on each bootstrap sample.

Algorithm 2: Bagged Decision Trees

Input: Training set $\mathcal{D}$ of size m; number of trees B.
Output: Ensemble $\{T_1, \ldots, T_B\}$.

1. **for** $b = 1$ to B:
   a. Draw a bootstrap sample $\mathcal{D}_b$ of size m from $\mathcal{D}$ (with replacement).
   b. Train a full (or depth-limited) decision tree T_b on $\mathcal{D}_b$.
2. **return** $\{T_1, \ldots, T_B\}$.

Hyperparameter B. Typical values are $B \in [64, 128]$. Increasing B never *hurts* accuracy, but the gains exhibit diminishing returns; setting $B \gg 128$ increases inference time without meaningful improvement.

7.3.5 Random Forests

Limitation of Bagging

In a bagged ensemble, all trees share access to the same full feature set. If one feature is a very strong predictor, nearly every tree will choose it as the root split, resulting in highly correlated trees. Correlated trees reduce the variance-reduction benefit of ensembling.

Random Feature Subsets

The **Random Forest** algorithm introduces a second source of randomness: at each node of each tree, rather than considering all n features, only a random subset of $k < n$ features is evaluated as candidates for the split.

$$k = \lfloor \sqrt{n} \rfloor \quad \text{(standard heuristic for classification).} \tag{7.7}$$

For regression tasks, $k = \lfloor n/3 \rfloor$ is often used instead.

Algorithm 3: Random Forest

Input: Training set $\mathcal{D}$, number of trees B, feature subset size k.
Output: Ensemble $\{T_1, \ldots, T_B\}$.

1. **for** $b = 1$ to B:
   a. Draw bootstrap sample $\mathcal{D}_b$ from $\mathcal{D}$.
   b. Grow tree T_b using modified splitting: at each node, randomly sample k features and select the best split from *only* those k features.
2. **return** $\{T_1, \ldots, T_B\}$.

Why Random Forests Work

- **Decorrelation.** Feature randomisation ensures that different trees use different splitting features, reducing the correlation between trees and thereby amplifying the variance-reduction effect of averaging.
- **Exploration.** Weaker features have a chance to be the best option in the restricted subset, so the ensemble explores a broader hypothesis space.
- **Variance reduction.** The ensemble variance is approximately σ^2_{tree}/B when trees are uncorrelated; feature randomisation brings real-world ensembles closer to this ideal.

7.3.6 Boosting and XGBoost

From Bagging to Boosting

Bagging trains trees *in parallel* on independent subsamples. **Boosting** trains trees *sequentially*: each new tree focuses on the examples that the current ensemble handles poorly.

The Boosting Intuition

The key idea behind boosting can be compared to deliberate practice: rather than repeatedly practising what one already does well, a skilled learner concentrates effort on areas of weakness. Analogously, in boosting:

- After each tree is added, the algorithm identifies which training examples were *misclassified* (or had high residual error).
- The *next* tree is trained on a reweighted (or resampled) version of the data that emphasises these difficult examples.

The Boosting Algorithm

Algorithm 4: Boosted Decision Trees

Input: Training set $\mathcal{D}$ of size m; number of trees B.
Output: Weighted ensemble $\{(T_b, \alpha_b)\}_{b=1}^{B}$.

1. Initialise uniform sample weights: $w_i = 1/m$ for all i.
2. **for** $b = 1$ to B:
   a. Train decision tree T_b on $\mathcal{D}$ weighted by $\{w_i\}$.
   b. Compute tree weight α_b (higher for more accurate trees).
   c. Update sample weights: increase w_i for misclassified examples, decrease for correctly classified ones.
   d. Normalise $\{w_i\}$ so they sum to 1.
3. **return** $\{(T_b, \alpha_b)\}$.

XGBoost

XGBoost (*eXtreme Gradient Boosting*) is the dominant open-source implementation of boosted trees. It frames boosting as gradient descent in function space: each new tree is fit to the *pseudo-residuals* (negative gradients of the loss function) of the current ensemble. Its key engineering and algorithmic features include:

- **Regularisation.** An explicit L_1 and L_2 penalty on leaf weights is built into the objective function, substantially reducing overfitting compared to classical AdaBoost.
- **Efficient computation.** XGBoost uses a quantile sketch algorithm to find near-optimal split thresholds in $O(n \log n)$ time and exploits CPU cache structure to accelerate memory access.
- **Sample weights.** Rather than literal resampling, XGBoost assigns a continuous weight to each training example, which is more stable and computationally efficient.
- **Sparse awareness.** Missing feature values are handled natively via a learned default direction at each split.
- **Competition track record.** XGBoost and its derivatives consistently win machine learning competitions on structured data, frequently alongside deep learning architectures.

Python Interface

XGBoost provides a concise, scikit-learn-compatible interface:

```
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)
y_pred_reg = reg.predict(X_test)
```

Listing 7.1. XGBoost for classification and regression.

7.3.7 Decision Trees vs. Neural Networks

Strengths of Decision Trees and Ensembles

Decision trees and ensemble methods excel in the following settings:

- **Tabular (structured) data.** When the input is a spreadsheet-like matrix of numeric and categorical features, tree ensembles are typically the fastest route to a high-performing model.
- **Fast training.** Tree ensembles train orders of magnitude faster than large neural networks, enabling rapid iteration and hyperparameter search.
- **Interpretability.** A single, shallow decision tree can be visualised and understood by domain experts, aiding debugging and regulatory compliance. (Large ensembles lose this property and behave as black boxes.)
- **No feature scaling required.** Tree splits are invariant to monotone transformations of the features; standardisation or normalisation is unnecessary.

Strengths of Neural Networks

- **Unstructured data.** Neural networks are the state of the art for images, audio, video, and raw text — data types for which tree-based models perform poorly.
- **Transfer learning.** Pre-trained neural networks can be fine-tuned on small task-specific datasets, a capability with no direct analogue in tree models.
- **End-to-end training.** Multiple neural network components can be jointly optimised via gradient descent, simplifying pipelines that would otherwise require separate training stages.
- **Mixed-modality inputs.** A single neural architecture can ingest both structured features and unstructured content (e.g., a product's tabular attributes *and* its image).

Comparative Summary

Table 7.3. Decision trees / ensembles vs. neural networks: when to use which.

Criterion	**Decision Trees / Ensembles**	**Neural Networks**
Primary data type	Tabular / structured	Unstructured, structured, mixed
Training speed	Fast	Slow (large models)
Interpretability	High (single trees)	Low (black box)
Transfer learning	Not available	Yes (pre-training + fine-tuning)
Recommended toolkit	XGBoost, Random Forest	PyTorch, TensorFlow, JAX

Practical Guidance

- **Use XGBoost or Random Forest** when your data fits neatly into a table, when you need fast iteration, or when interpretability is a requirement.
- **Use a neural network** when your task involves images, text, audio, video, or any combination of modalities; or when the task benefits strongly from transfer learning.

7.3.8 Chapter Summary

This chapter has introduced the complete decision tree framework, from individual trees to state-of-the-art ensemble methods.

Part I — Decision Trees presented the basic model: a hierarchical structure of decision and leaf nodes that classifies new examples through top-down traversal. Features may be binary, multi-valued categorical (handled via one-hot encoding), or continuous.

Part II — Decision Tree Learning described how trees are built. The recursive algorithm selects, at each node, the feature that maximises *Information Gain* — the reduction in entropy (Equation 7.2) — and recurses on the resulting subsets. Stopping criteria (maximum depth, minimum examples, IG threshold) control overfitting. Regression trees replace entropy with variance reduction (Equation 7.5) and predict the mean of leaf-node training targets.

Part III — Tree Ensembles addressed the instability of single trees by combining many trees trained on bootstrapped subsamples (Bagging). The Random Forest algorithm additionally randomises the feature subset considered at each split, decorrelating trees and further reducing variance. Boosting — exemplified by XGBoost — takes a complementary sequential approach, focusing successive trees on the training examples that the current ensemble finds most difficult, with built-in regularisation to prevent overfitting.

CHAPTER 8

Unsupervised Learning

Chapter Overview. Unsupervised learning extracts structure from unlabelled data. This chapter covers three major algorithms: K-means clustering (with the distortion objective, initialisation strategies, and elbow method), anomaly detection via Gaussian density estimation, and dimensionality reduction using principal component analysis (PCA).

8.1 Clustering

8.1.1 Overview and Motivation

Clustering is one of the foundational algorithms in *unsupervised learning*. Its goal is to automatically discover patterns, groupings, or structures within a dataset—without any human-provided labels or guidance. Given a collection of data points, a clustering algorithm partitions them into coherent subsets called *clusters*, such that points within the same cluster are more similar to one another than to points in different clusters.

Clustering arises naturally in a wide range of scientific and commercial settings. Consider the challenge of analysing millions of customer transactions, classifying celestial objects from telescope imagery, or grouping news articles by topic. In each case, labelled training data would be prohibitively expensive or simply unavailable. Clustering provides a principled way to extract meaningful structure without supervision.

Definition 8.1.1 (Cluster)**.** A *cluster* is a subset $\mathcal{C} \subseteq \{x^{(1)}, \ldots, x^{(m)}\}$ of training examples such that points in $\mathcal{C}$ are closer to one another—under some chosen distance metric—than they are to points outside $\mathcal{C}$.

8.1.2 Supervised vs. Unsupervised Learning

Understanding the distinction between supervised and unsupervised learning is essential before studying clustering algorithms in depth.

Supervised Learning

In supervised learning, the training set consists of input–output pairs:

$$\mathcal{D}_{\text{sup}} = \big\{(x^{(1)}, y^{(1)}),\, (x^{(2)}, y^{(2)}),\, \ldots,\, (x^{(m)}, y^{(m)})\big\}.$$

Each example $x^{(i)} \in \mathbb{R}^n$ is paired with a target label $y^{(i)}$. The model learns a mapping $f : \mathbb{R}^n \to \mathcal{Y}$ by minimising a loss between predictions $\hat{y}^{(i)} = f(x^{(i)})$ and true labels $y^{(i)}$. For example, in binary classification the model learns a decision boundary that separates two classes.

Unsupervised Learning

In unsupervised learning, only the input features are available:

$$\mathcal{D}_{\text{unsup}} = \{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}.$$

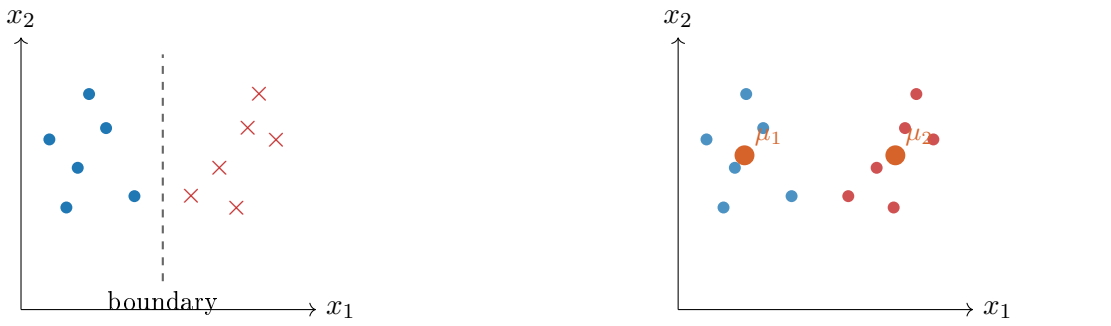
There are no target labels $y^{(i)}$. The algorithm cannot be told whether its predictions are "correct" in the classical sense. Instead, it seeks to uncover *latent structure*—interesting groupings, manifolds, or patterns—inherent in the data itself.

Table 8.1 summarises the key differences.

Table 8.1. Supervised learning vs. unsupervised learning (clustering).

Feature	Supervised Learning	Unsupervised (Clustering)
Labels (y)	Required	Not provided
Data format	(x, y) pairs	x only
Primary goal	Classification / Regression	Finding structure / groups
Visual output	Decision boundary	Cluster centroids / groups
Error signal	Labelled ground truth	Cost / distortion function

Figure 8.1 illustrates this contrast visually.



(a) Supervised: a decision boundary separates labelled classes.

(b) Unsupervised: clusters and centroids are discovered from unlabelled data.

Figure 8.1. Conceptual comparison of supervised and unsupervised learning.

8.1.3 Real-World Applications of Clustering

Clustering is a highly versatile tool employed across numerous industries:

1. **News aggregation.** Grouping news articles by topic so that users receive an organised feed (e.g., all articles about a particular event or subject appear together).
2. **Market segmentation.** Identifying distinct customer groups based on purchasing behaviour, demographics, or motivations. For example, learners on an online platform might be segmented into 'skill growth,"' career development," and "AI research" categories.
3. **Genomics and DNA analysis.** Grouping individuals according to genetic expression data, enabling researchers to identify people with similar biological traits, disease predispositions, or responses to treatment.
4. **Astronomy.** Analysing large-scale observational data to group celestial bodies into galaxies, star clusters, or other coherent cosmic structures.
5. **Image compression.** Representing an image using only K representative colours (centroids), reducing file size while preserving visual quality.

8.1.4 The K-Means Algorithm

The most widely used clustering algorithm is **K-means**. It partitions a dataset of m examples into K non-overlapping clusters by iteratively alternating between two steps: *cluster assignment* and *centroid update*.

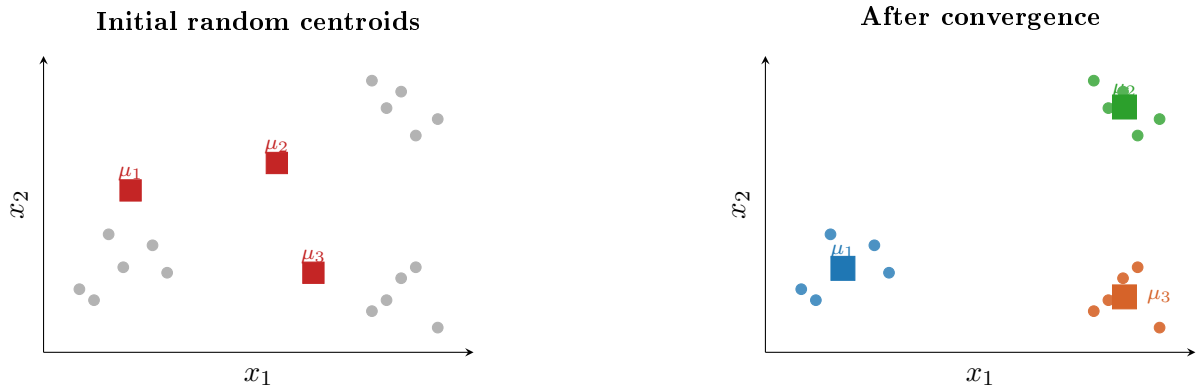


Figure 8.2. K-means with $K = 3$. **Left:** three centroids (squares) are placed randomly. **Right:** after assignment and update iterations, the centroids converge to the true cluster centres and the data is cleanly partitioned.

Initialisation

1. Choose the number of clusters $K < m$.
2. Randomly select K distinct training examples and place the initial cluster centroids $\mu_1, \mu_2, \ldots, \mu_K \in \mathbb{R}^n$ at those locations. Using actual data points (rather than arbitrary positions in space) ensures the centroids begin inside the data distribution.

Iterative Steps

The following two steps are repeated until convergence (i.e., until no assignments change):

Step 1 — Cluster Assignment.. For each training example $x^{(i)}$, assign it to the nearest centroid:

$$c^{(i)} := \underset{k \in \{1,\ldots,K\}}{\arg\min} \left\| x^{(i)} - \mu_k \right\|^2, \quad i = 1, \ldots, m.$$

Here $c^{(i)} \in \{1, \ldots, K\}$ denotes the index of the cluster to which example i is assigned, and $\|\cdot\|^2$ is the squared Euclidean (L2) norm. Using the squared distance is computationally equivalent to using the distance itself for the purpose of minimisation, while avoiding the cost of a square-root operation.

Step 2 — Centroid Update.. For each cluster k, recompute its centroid as the arithmetic mean of all examples currently assigned to it:

$$\mu_k := \frac{1}{|\mathcal{C} * k|} \sum * i : \; c^{(i)} = k x^{(i)},$$

where $\mathcal{C}_k = \{i : c^{(i)} = k\}$ is the set of indices assigned to cluster k.

If at some iteration $\mathcal{C}_k = \emptyset$ (no examples assigned to cluster k), the standard convention is to *eliminate* that cluster, reducing K by one. An alternative is to reinitialise the centroid randomly; however, elimination is more common in practice.

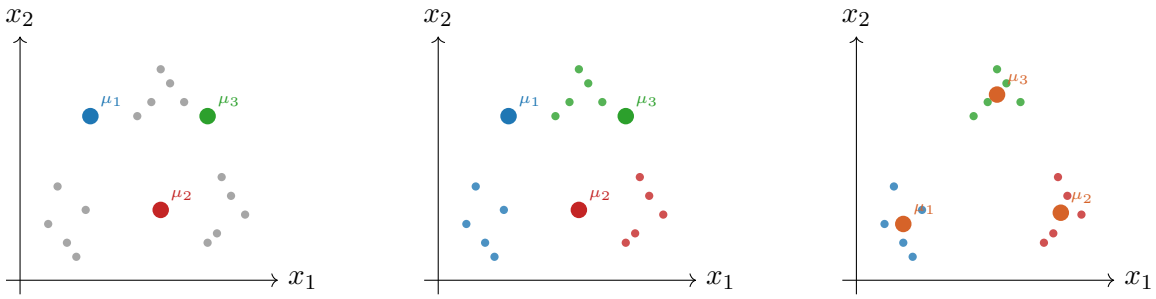
Algorithm 1 summarises the complete procedure.

Algorithm 1 K-Means Clustering

Require: Dataset $\{x^{(1)}, \ldots, x^{(m)}\}$, number of clusters K

1: Randomly pick K examples; set $\mu_1, \ldots, \mu_K$ to these examples
2: **repeat**
3: **for** $i = 1$ to m **do** ▷ Cluster assignment
4: $c^{(i)} \leftarrow \arg\min_k \|x^{(i)} - \mu_k\|^2$
5: **end for**
6: **for** $k = 1$ to K **do** ▷ Centroid update
7: $\mu_k \leftarrow \frac{1}{|\mathcal{C} * k|} \sum * i : \; c^{(i)} = k x^{(i)}$
8: **end for**
9: **until** assignments $c^{(1)}, \ldots, c^{(m)}$ do not change

Figure 8.3 illustrates three snapshots of K-means on a two-dimensional toy dataset.



(a) Initialisation **(b)** After assignment step **(c)** After convergence

Figure 8.3. K-means on a two-dimensional dataset with $K = 3$. Coloured circles are data points assigned to their respective cluster; orange filled circles are the centroids.

Application to Non-Separated Data

K-means performs well even when clusters are not obviously separated. A canonical example is **T-shirt sizing**:

Example 8.1.1 (T-shirt sizing)**.** A clothing manufacturer records the height and weight of a large population. The data form a continuous, overlapping cloud with no clear gaps. Running K-means with $K = 3$ yields centroids corresponding to 'Small,"' Medium," and "Large" body types. The centroid coordinates directly inform the manufacturer's target dimensions for each size, maximising fit across the population.

8.1.5 The K-Means Optimisation Objective

Mathematical Notation

We introduce the following notation:

- $c^{(i)} \in \{1, \ldots, K\}$: the cluster index assigned to example $x^{(i)}$.
- $\mu_k \in \mathbb{R}^n$: the centroid of cluster k.
- $\mu_{c^{(i)}}$: the centroid of the cluster to which $x^{(i)}$ is assigned.

For instance, if $x^{(10)}$ is assigned to cluster 2 (i.e., $c^{(10)} = 2$), then $\mu_{c^{(10)}} = \mu_2$.

The Distortion (Cost) Function

K-means minimises the **distortion function** J, defined as the mean squared distance from each example to its assigned centroid:

$$J\Big(c^{(1)}, \ldots, c^{(m)}, \mu_1, \ldots, \mu_K\Big) \;=\; \frac{1}{m}\sum_{i=1}^{m} \big\| x^{(i)} - \mu_{c^{(i)}} \big\|^2.$$

The optimisation problem is:

$$\min_{\substack{c^{(1)},\ldots,c^{(m)} \\ \mu_1,\ldots,\mu_K}} J\Big(c^{(1)},\ldots,c^{(m)},\,\mu_1,\ldots,\mu_K\Big)\,.$$

How Each Step Reduces J

Although K-means does not use gradient descent, its two steps are a coordinate-descent procedure that monotonically decreases J:

Step 1: **Cluster assignment** holds $\mu_1,\ldots,\mu_K$ fixed and minimises J with respect to $c^{(1)},\ldots,c^{(m)}$. For each point $x^{(i)}$, assigning it to the nearest centroid minimises $\|x^{(i)}-\mu_{c^{(i)}}\|^2$ individually, hence minimises the sum.

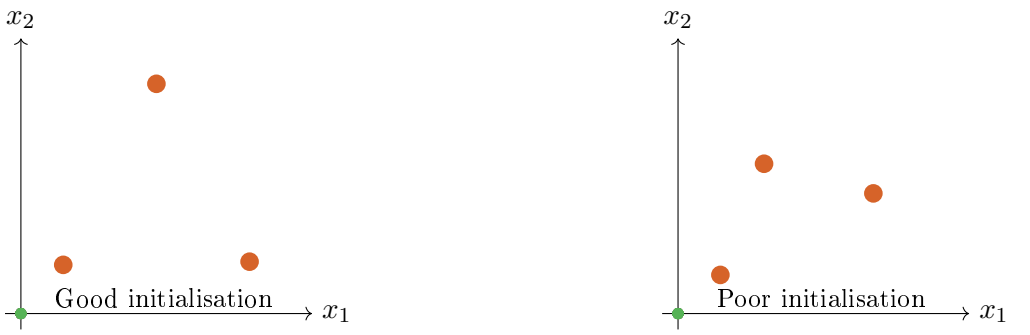
Step 2: **Centroid update** holds $c^{(1)},\ldots,c^{(m)}$ fixed and minimises J with respect to $\mu_1,\ldots,\mu_K$. For a fixed cluster k, the value of μ_k that minimises $\sum_{i:c^{(i)}=k}\|x^{(i)}-\mu_k\|^2$ is the arithmetic mean of those points—which is precisely the update rule.

Remark 8.1.1 (Guaranteed convergence)**.** Because every step of the algorithm is guaranteed to decrease J or leave it unchanged, K-means always converges in a finite number of iterations. If J ever *increases*, this signals a bug in the implementation.

8.1.6 Initialisation and Local Optima

The Problem of Local Optima

The final clustering produced by K-means depends on the initial centroid positions. Poor initialisations can cause the algorithm to converge to a **local minimum** of J that does not capture the true structure of the data. Figure 8.4 illustrates two runs of K-means with $K=3$ on the same dataset—one converges to a good solution, the other to a poor one.



(a) Good solution (low J) **(b)** Local-optimal solution (high J)

Figure 8.4. Two different initialisations with $K=3$. The right panel shows a local minimum where natural clusters are incorrectly merged or split.

Multiple Random Initialisations

The standard remedy is to run K-means multiple times with different random initialisations and keep the solution with the lowest distortion:

1. For run $= 1$ to R (typically $R \in [50, 1000]$):
   a. Randomly pick K training examples; initialise centroids.
   b. Run K-means to convergence; record assignments $\{c^{(i)}\}$ and centroids $\{\mu_k\}$.
   c. Compute distortion J_{run}.
2. Select the run with the lowest J_{run} as the final solution.

Multiple restarts are especially important when K is small (e.g., $K \leq 10$); for large K, K-means is usually less sensitive to initialisation.

8.1.7 Choosing the Number of Clusters K

Selecting the correct value of K is inherently ambiguous in unsupervised learning because there is no ground-truth label against which to validate. Two principal strategies are used in practice.

The Elbow Method

Plot the distortion J as a function of K. In some datasets, the curve exhibits a sharp "elbow"—a point of rapidly diminishing returns—which can suggest a reasonable K.

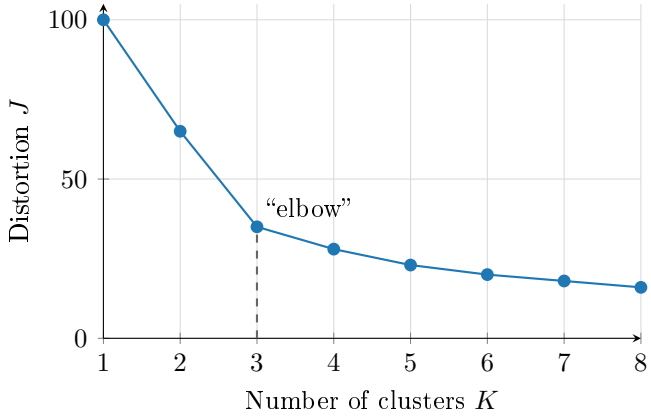


Figure 8.5. The elbow method: the curve bends sharply at $K = 3$, suggesting this as a candidate number of clusters.

Remark 8.1.2 (Limitation of the elbow method)**.** In many real-world datasets the distortion curve decreases smoothly without a clear kink, making the elbow method unreliable. Moreover, one should *never* choose K simply to minimise J, because J will always decrease as K increases, reaching $J = 0$ when $K = m$.

Choosing K Based on Downstream Purpose

The most principled approach is to select K according to the *ultimate objective* of the clustering task:

Example 8.1.2 (T-shirt sizing revisited)**.** With $K = 3$ (S, M, L), manufacturing is simpler but fit is coarser. With $K = 5$ (XS, S, M, L, XL), customers are better served but production costs increase. The optimal K is determined by the business trade-off between cost and customer satisfaction—not by J alone.

Example 8.1.3 (Image compression)**.** Using K colours to represent an image: a larger K yields higher quality but larger file size. The choice of K depends on the desired quality–compression trade-off.

8.2 Anomaly Detection

8.2.1 Introduction and Motivation

Anomaly detection (also called *outlier detection*) is the task of identifying data points that deviate significantly from the expected, "normal" pattern. Unlike most supervised classifiers, an anomaly detection system is primarily trained on *normal* examples; the goal is to assign a low probability score to any future example that looks unusual.

Definition 8.2.1 (Anomaly)**.** An *anomaly* (or outlier) is an observation x_{test} whose probability $p(x_{\text{test}})$ under a model fitted on normal data falls below a chosen threshold ϵ:

$$x_{\text{test}} \text{ is anomalous} \iff p(x_{\text{test}}) < \epsilon.$$

Anomaly detection enables early intervention, enhances system stability, and helps organisations maintain control and safety across a wide range of processes.

8.2.2 Types of Anomalies

Three canonical categories of anomalies are distinguished in the literature.

Point Anomalies

A single data point that differs significantly from the overall distribution.

Example 8.2.1. A credit card transaction of \$15,000 for a customer whose typical monthly spend is \$500.

Contextual Anomalies

A data point that is normal in absolute terms but abnormal given its context.

Example 8.2.2. A temperature reading of 85°F (29°C) is unremarkable in July but anomalous in January in New York.

Collective Anomalies

A group of data points, each individually normal, that are anomalous when considered together.

Example 8.2.3. Thousands of network packets arriving from a single IP address over a few seconds—each packet is normal in isolation, but the burst collectively suggests a denial-of-service (DoS) attack.

8.2.3 Real-World Applications

Anomaly detection is employed across a broad range of domains.

1. **Fraud detection.** Financial institutions monitor transactions for unusual amounts, foreign locations, or abnormal sequences. Accounts that deviate from established behaviour are flagged for manual review or identity verification (e.g., SMS confirmation).
2. **Cybersecurity.** Network traffic is monitored for deviations indicative of DoS attacks, phishing, or malware propagation. Anomalous login patterns or CPU usage can signal unauthorised access.
3. **Manufacturing quality control.** Performance signatures of products (e.g., aircraft engine vibration and heat output) are compared to the normal distribution; outliers are removed from the production line.
4. **Healthcare monitoring.** Wearable devices detect unusual vital signs (heart rate, blood pressure) and notify caregivers in real time.
5. **Data centre monitoring.** Features such as memory usage, disk I/O, CPU load, and CPU-to-network-traffic ratios are tracked to detect failing hardware or potential intrusions.

8.2.4 The Density Estimation Approach

The dominant framework for anomaly detection is **density estimation**: fit a probabilistic model $p(x)$ to the training data, and flag a test point x_{test} as anomalous whenever $p(x_{\text{test}})$ is very small.

Decision Rule

1. Build model $p(x)$ from a training set $\{x^{(1)}, \ldots, x^{(m)}\}$ of normal examples.
2. Choose a threshold $\epsilon > 0$ (small).

3. For a new test point:

$$\text{Decision} = \begin{cases} \text{Anomaly} & \text{if } p(x_{\text{test}}) < \epsilon, \\ \text{Normal} & \text{if } p(x_{\text{test}}) \geq \epsilon. \end{cases}$$

Figure 8.6 illustrates this idea in one dimension.

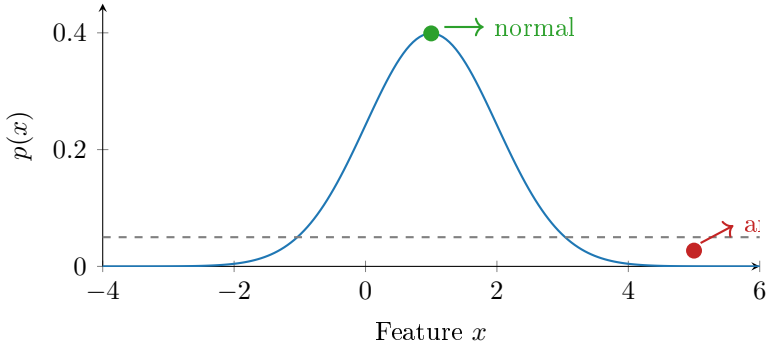


Figure 8.6. Density estimation for anomaly detection. Points with $p(x) < \epsilon$ (below the dashed line) are flagged as anomalies.

8.2.5 The Gaussian (Normal) Distribution

The most common parametric form used for $p(x)$ is the **Gaussian distribution**, often called the "bell curve."

Definition and Parameters

If a random variable X follows a Gaussian distribution with mean μ and variance σ^2, we write $X \sim \mathcal{N}(\mu, \sigma^2)$, and its probability density function is:

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

Key properties:

- The curve is symmetric about μ and attains its maximum at $x = \mu$.
- The parameter σ (standard deviation) controls the spread: small σ produces a narrow, tall curve; large σ produces a wide, flat curve.
- The total area under the curve is always 1: $\int_{-\infty}^{\infty} p(x;\, \mu, \sigma^2)\, dx = 1$.

Figure 8.7 shows Gaussian curves for several parameter combinations.

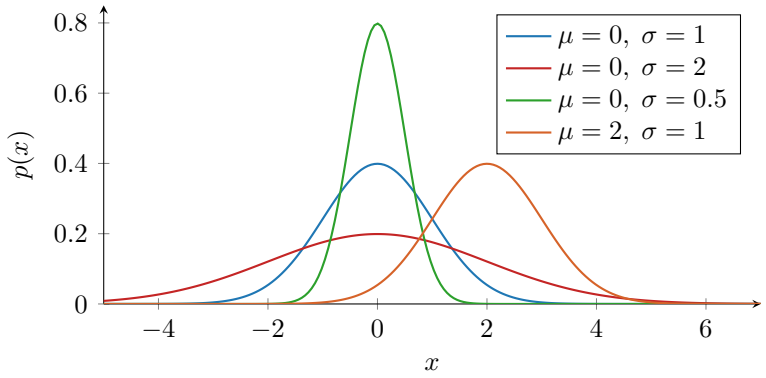


Figure 8.7. Gaussian distributions for various μ and σ values. Shifting μ translates the curve; changing σ alters width and height.

Parameter Estimation (Maximum Likelihood)

Given a training set $\{x^{(1)}, \ldots, x^{(m)}\}$ of scalar observations, the maximum likelihood estimates of μ and σ^2 are:

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

Remark 8.2.1. Some statistics textbooks use a denominator of $m-1$ (Bessel's correction) to obtain an unbiased estimator. In machine learning practice, m is standard and performs effectively on large datasets.

8.2.6 Multi-Dimensional Anomaly Detection Algorithm

In practice, each example $x^{(i)} \in \mathbb{R}^n$ has n features. We model each feature independently with its own Gaussian.

Density Model

Under the **independence assumption** (features are statistically independent), the joint density factors as:

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

where

$$p(x_j;\, \mu_j, \sigma_j^2) = \frac{1}{\sqrt{2\pi}\,\sigma_j} \exp\left(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2}\right).$$

The independence assumption is technically restrictive, but in practice K-means performs well even when features are mildly correlated.

Algorithm Steps

Step 1: Feature selection. Choose n features $x_1, \ldots, x_n$ that are informative about the type of anomaly to detect.

Step 2: Parameter fitting. For each feature $j = 1, \ldots, n$, compute:

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}, \qquad \sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x_j^{(i)} - \mu_j\right)^2.$$

In vectorised form: $\vec{\mu} = \frac{1}{m}\sum_{i=1}^{m} \vec{x}^{(i)}$.

Step 3: Prediction. For a new test example $\vec{x} * \text{test}$, compute:

$$p(\vec{x} * \text{test}) = \prod_{j=1}^{n} p\left(x_{\text{test},j};\, \mu_j, \sigma_j^2\right).$$

Flag as anomaly if $p(\vec{x}_{\text{test}}) < \epsilon$.

The algorithm is summarised in Algorithm 2.

Algorithm 2 Multi-Dimensional Anomaly Detection

Require: Training set $\{x^{(1)}, \ldots, x^{(m)}\}$, threshold ϵ

for $j = 1$ to n **do**
  $\mu_j \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$
  $\sigma_j^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2$
end for
procedure PREDICT($\vec{x} * \text{test}$)
  $p \leftarrow \prod * j = 1^n \frac{1}{\sqrt{2\pi}\sigma_j} \exp\left(-\frac{(x_{\text{test},j} - \mu_j)^2}{2\sigma_j^2}\right)$
  if $p < \epsilon$ **then return anomaly**
  else return normal
  end if
end procedure

8.2.7 Developing and Evaluating an Anomaly Detection System

Dataset Split

Although anomaly detection is primarily unsupervised, having a small number of labelled anomalies is invaluable for tuning and evaluation. A typical split for a dataset with 10,000 normal and 20 anomalous examples is:

Table 8.2. Dataset split for anomaly detection development.

Set	Normal ($y = 0$)	Anomalous ($y = 1$)	Purpose
Training	6,000	0	Fit $p(x)$ parameters
Cross-Validation	2,000	10	Tune ϵ, select features
Test	2,000	10	Final performance evaluation

Evaluation Metrics for Skewed Data

Because anomalies are rare, the dataset is highly skewed (e.g., 99.9% normal). Standard accuracy is therefore misleading: a classifier that labels *everything* as normal would achieve 99.9% accuracy while detecting zero anomalies. Instead, the following metrics are preferred:

- **Precision:** $\dfrac{\text{TP}}{\text{TP} + \text{FP}}$ — of all flagged anomalies, what fraction are true anomalies?
- **Recall:** $\dfrac{\text{TP}}{\text{TP} + \text{FN}}$ — of all true anomalies, what fraction did we detect?
- **F_1-score:** $\dfrac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$ — harmonic mean combining precision and recall into a single scalar.

The threshold ϵ is chosen to *maximise the F_1-score* on the cross-validation set.

8.2.8 Anomaly Detection vs. Supervised Learning

Table 8.3 captures the key decision criteria for choosing between the two paradigms.

Table 8.3. Anomaly detection vs. supervised learning.

Feature	Anomaly Detection	Supervised Learning
Positive examples	Very few (0–20 common)	Many
Negative examples	Many	Many
Anomaly types	Diverse, unpredictable; future anomalies may look nothing like past ones	Recurring; future positives resemble training examples
Key use case	Fraud, novel defects, novel intrusions	Spam, known defects, weather

The fundamental distinction is about *generalisation*: anomaly detection is favoured when the set of possible anomaly patterns is open-ended and constantly evolving, whereas supervised learning is preferred when anomaly patterns are stable and well-represented in the training data.

8.2.9 Feature Engineering for Anomaly Detection

Gaussian Transformation

The density model assumes each feature follows a Gaussian distribution. If a feature's empirical histogram is heavily skewed, the model fit will be poor. Common transformations to induce approximate normality include:

- **Log transformation:** $x \leftarrow \log(x)$ or $x \leftarrow \log(x + c)$ for some constant $c > 0$ (to avoid $\log(0)$).
- **Power transformation:** $x \leftarrow x^{1/2}$, $x^{1/3}$, or more generally x^p for $0 < p < 1$.

Figure 8.8 illustrates how a log transformation can convert a right-skewed distribution into an approximately Gaussian one.

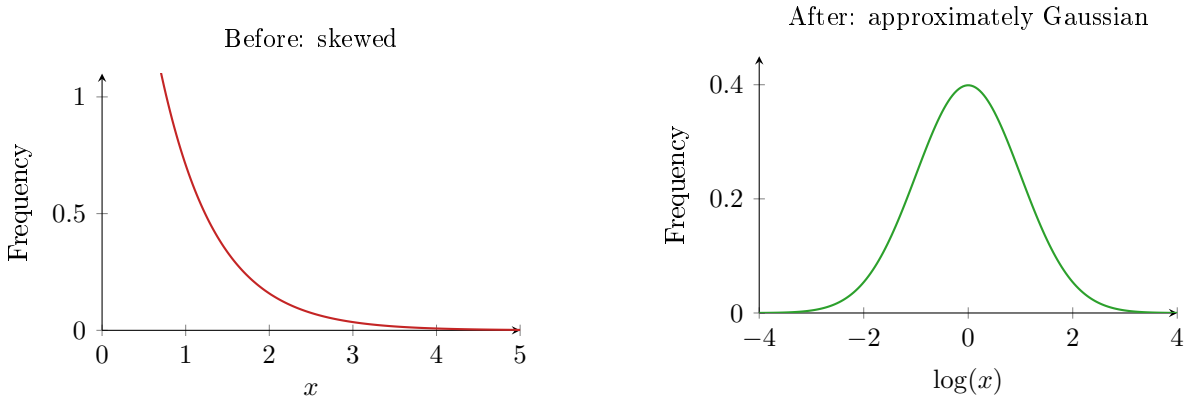


Figure 8.8. Log transformation of a right-skewed feature (left) to an approximately Gaussian distribution (right).

Error Analysis and Feature Creation

If a known anomalous example receives a high probability score $p(x) \geq \epsilon$ (a false negative), perform the following **error analysis**:

1. Examine the missed anomaly; identify what physical characteristic makes it unusual.
2. Design a new feature x_{n+1} that captures this characteristic.
3. Verify that the new feature takes an extreme value for anomalies while remaining moderate for normal examples.

Example 8.2.4 (Data centre monitoring)**.** Let x_1 = CPU load and x_2 = network traffic. A machine stuck in an infinite loop may exhibit high CPU load but low network traffic—individually, each reading looks normal. The ratio feature

$$x_3 = \frac{x_1}{x_2}$$

takes an extreme value for this pathology and allows the algorithm to flag it.

Summary of the Development Cycle

A practical anomaly detection system is built through the following iterative workflow:

1. Fit $p(x)$ using only the "normal" training examples.
2. Evaluate performance on the cross-validation set; identify missed anomalies.
3. Perform error analysis: inspect false negatives and false positives.
4. Add or transform features to improve separation between normal and anomalous examples.
5. Adjust the threshold ϵ to optimise the F_1-score.
6. Repeat until performance is satisfactory; report final metrics on the held-out test set.

8.2.10 Anomaly Detection Techniques: A Broader View

Beyond the Gaussian density model, a rich family of techniques is available for anomaly detection.

Statistical Methods

Compute summary statistics (mean μ, standard deviation σ, z-scores) and flag data points that lie beyond a chosen number of standard deviations from the mean.

Density-Based Methods

Assess the local density of data points. Algorithms such as **DBSCAN** treat points in low-density regions as anomalies. The **Local Outlier Factor (LOF)** compares the local density of a point to those of its neighbours.

Distance-Based Methods

Measure distances between examples. The **k-nearest-neighbour** (k-NN) approach flags points that are far from their k closest neighbours as outliers.

Ensemble Methods

Combine multiple anomaly detectors to improve robustness. **Isolation Forest** is a popular ensemble method that isolates anomalies by constructing random trees: anomalous points are isolated close to the root and require fewer splits than normal points.

Deep Learning Methods

Autoencoders are neural networks trained to reconstruct their input. Trained on normal data, they learn a compact representation of normality. An anomalous input is poorly reconstructed, yielding a high reconstruction error—which serves as the anomaly score.

Learning Paradigms for Anomaly Detection

Table 8.4 classifies the three learning paradigms applicable to anomaly detection.

Table 8.4. Learning paradigms for anomaly detection.

Paradigm	Labels required	Key methods	Limitations
Supervised	Both classes labelled	Decision trees, SVM, neural nets	Requires costly labels; may miss novel anomalies
Unsupervised	None	Clustering, PCA, autoencoders	May confuse subtle anomalies with normal points
Semi-supervised	Normal class only	Reconstruction-error neural nets	Depends on quality of normal-class representation

8.2.11 Anomaly Detection in Advanced Systems

Modern production systems augment anomaly detection with additional infrastructure:

- **Real-time traffic analysis.** Advanced load balancers continuously monitor traffic patterns and detect unusual spikes or drops that may indicate attacks or hardware failures.
- **Rate limiting.** Automatically throttles requests from suspicious sources, reducing the impact of anomalous traffic.
- **Adaptive load balancing.** Uses AI algorithms to respond to changing traffic patterns and identify deviations from expected behaviour.
- **Integration with security frameworks.** Correlates anomaly signals with broader security events, enhancing threat detection across the full system stack.
- **Feedback loops.** Verified outcomes (anomalous or normal) are fed back into the model to continually refine its accuracy over time.

Chapter Summary

This chapter introduced the two principal families of unsupervised learning algorithms:

Clustering (Section 1).. Clustering discovers groups of similar data points without labels. The **K-means algorithm** alternates between assigning each point to its nearest centroid and recomputing the centroid as the mean of assigned points. This process minimises the **distortion**

function

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

K-means is sensitive to initialisation and may converge to local minima; running multiple random initialisations and keeping the lowest-cost solution is standard practice. The number of clusters K is best chosen based on the downstream purpose of the clustering rather than purely on J.

Anomaly Detection (Section 2).. Anomaly detection builds a probabilistic model $p(x)$ from normal training data and flags test points with $p(x) < \epsilon$ as anomalies. The most common model assumes Gaussian features:

$$p(\vec{x}) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\,\sigma_j} \exp\left(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2}\right).$$

Evaluation relies on precision, recall, and the F_1-score rather than accuracy, because anomalies are rare and datasets are skewed. Feature engineering—including log/power transformations and ratio features—is crucial to ensure the algorithm captures subtle anomalies that escape detection under the raw features. The choice between anomaly detection and supervised learning hinges on the number of labelled anomalies available and the diversity of anomaly types expected in deployment.

CHAPTER 9

Recommender Systems

Chapter Overview. Recommender systems predict user preferences and power platforms like Netflix, Spotify, and Amazon. This chapter develops collaborative filtering (memory-based and matrix-factorisation variants), content-based filtering, and the deep learning retrieval–ranking architecture that combines the strengths of both approaches.

Introduction

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

Formally, a recommender system operates on a set of n users $\mathcal{U} = \{u_1, \ldots, u_n\}$ and a set of m items $\mathcal{I} = \{i_1, \ldots, i_m\}$. The observed interactions are summarised in a rating matrix $\mathbf{R} \in \mathbb{R}^{n \times m}$, where entry r_{ui} denotes the rating that user u gave to item i. Because most users rate only a tiny fraction of all available items, this matrix is extremely *sparse*: a typical platform may have 99% or more of its entries unobserved (denoted "?"). The task of a recommender system is to infer the missing entries and rank items by their predicted relevance to each user.

Figure 9.1 illustrates a small example of such a matrix.

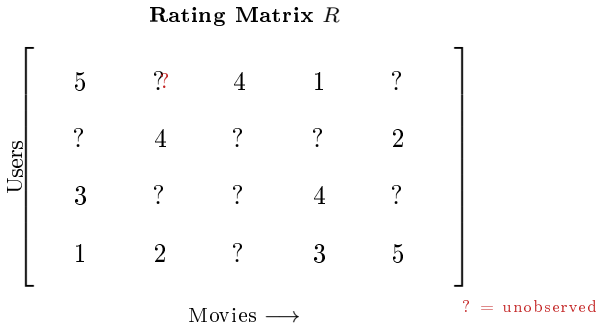


Figure 9.1. A user–item rating matrix. Orange cells marked "?" represent missing ratings that the recommender system is trained to predict.

This chapter covers three major paradigms for building recommender systems:

1. **Collaborative Filtering (CF)**: exploits the collective behaviour of all users to recommend items, based on the idea that users who agreed in the past tend to agree in the

future.

2. **Content-Based Filtering**: recommends items similar to those a user has previously liked, relying on item feature descriptions rather than other users' ratings.
3. **Principal Component Analysis (PCA)**: a dimensionality reduction technique widely used as a pre-processing step in recommender systems to uncover latent structure and reduce noise.

9.1 Collaborative Filtering

9.1.1 Core Idea

Collaborative filtering is the most widely studied family of recommendation algorithms. Its fundamental assumption is the *collaborative* premise: users with similar tastes in the past will share similar tastes in the future. No item content information is required — the system learns entirely from the pattern of ratings.

Given a partially observed rating matrix $\mathbf{R}$, the goal is to estimate $\hat{r} * ui$ for every user u and item i pair where $r * ui$ is unknown. Recommendations are generated by ranking items by their predicted scores for each user.

9.1.2 Memory-Based Collaborative Filtering

Memory-based (or neighbourhood-based) methods directly use the observed ratings to compute similarities between users or items. They require no explicit model training and are therefore easy to understand and update.

User-Based Collaborative Filtering

The intuition is: *"Find users similar to the target user; use their ratings to predict what the target user would rate."* Concretely, to predict r_{ui} for a target user u and item i:

1. Compute similarity $\text{sim}(u, v)$ between user u and all other users v who have rated item i.
2. Select the k most similar users (the *neighbourhood*).
3. Predict the rating as a weighted average of the neighbours' ratings, possibly adjusted for each user's average:

$$\hat{r} * ui = \bar{r}u + \frac{\sum v \in \mathcal{N}(u)\text{sim}(u, v)\,(r * vi - \bar{r} * v)}{\sum * v \in \mathcal{N}(u)\,|\text{sim}(u, v)|} \tag{9.1}$$

where $\bar{r}_u$ is the mean rating of user u and $\mathcal{N}(u)$ denotes the k-nearest neighbours of u.

Item-Based Collaborative Filtering

Symmetrically, item-based CF computes similarity between items rather than between users. The prediction for user u on item i is:

$$\hat{r} * ui = \frac{\sum * j \in \mathcal{N}(i) \text{sim}(i, j)\, r_{uj}}{\sum_{j \in \mathcal{N}(i)} |\text{sim}(i, j)|} \tag{9.2}$$

where $\mathcal{N}(i)$ is the set of items similar to i that user u has already rated.

Figure 9.2 illustrates the difference between the two neighbourhood strategies.

[Illustration] — see text description

Figure 9.2. Left: user-based CF highlights similar users (rows). Right: item-based CF highlights similar items (columns). Red/blue borders indicate the neighbourhood used for prediction.

Similarity Measures

Several similarity metrics are used in practice.

Cosine Similarity.. Treats each user (or item) as a vector of ratings and measures the angle between them:

$$\text{sim}_{\cos}(\boldsymbol{a}, \boldsymbol{b}) = \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{\|\boldsymbol{a}\|_2 \, \|\boldsymbol{b}\|_2} = \frac{\sum_k a_k b_k}{\sqrt{\sum_k a_k^2}\sqrt{\sum_k b_k^2}} \tag{9.3}$$

The result lies in $[-1, 1]$; higher values indicate more similar taste profiles. Figure 9.3 shows three user vectors in a two-dimensional feature space. Users A and B are close in angle (high cosine similarity), while A and C are further apart.

[Illustration] — see text description

Figure 9.3. Cosine similarity between users A, B, and C in a two-dimensional genre-rating space. Users A and B share a high cosine similarity, indicating compatible preferences.

Pearson Correlation.. Accounts for the fact that users have different rating scales (some users always rate high; others are generally harsh critics):

$$\text{sim} * \text{Pearson}(u, v) = \frac{\sum *i \in \mathcal{I} * uv(r * ui - \bar{r} * u)(r * vi - \bar{r} * v)}{\sqrt{\sum *i \in \mathcal{I} * uv(r * ui - \bar{r} * u)^2}\sqrt{\sum *i \in \mathcal{I} * uv(r * vi - \bar{r}_v)^2}} \tag{9.4}$$

where $\mathcal{I}_{uv}$ is the set of items rated by both users u and v.

Adjusted Cosine (for Item-Based CF).. When computing item similarity, the Pearson formulation is often adapted as *adjusted cosine*, which subtracts the mean rating of each *user* (rather than each item) to correct for individual rating biases:

$$\text{sim} * \text{adj}(i, j) = \frac{\sum *u \in \mathcal{U} * ij(r * ui - \bar{r} * u)(r * uj - \bar{r} * u)}{\sqrt{\sum *u \in \mathcal{U} * ij(r * ui - \bar{r} * u)^2}\sqrt{\sum *u \in \mathcal{U} * ij(r * uj - \bar{r}_u)^2}} \tag{9.5}$$

9.1.3 Model-Based Collaborative Filtering: Matrix Factorization

Memory-based methods suffer from scalability and sparsity problems: computing pairwise user similarities over millions of users is expensive, and with very few ratings per user the similarity estimates are noisy. *Matrix Factorization* (MF) addresses these issues by learning a compact latent representation for every user and item.

Low-Rank Matrix Factorization

The key idea is to approximate the rating matrix $\mathbf{R} \in \mathbb{R}^{n \times m}$ as the product of two low-rank factor matrices:

$$\mathbf{R} \approx \mathbf{U}\,\mathbf{V}^{\top} \tag{9.6}$$

where $\mathbf{U} \in \mathbb{R}^{n \times k}$ is the *user factor matrix* and $\mathbf{V} \in \mathbb{R}^{m \times k}$ is the *item factor matrix*, with $k \ll \min(n, m)$ being the number of *latent factors*. Each row $\boldsymbol{u}_u \in \mathbb{R}^k$ encodes the preferences of user u in a k-dimensional latent space, and each row $\boldsymbol{v}_i \in \mathbb{R}^k$ encodes the characteristics of item i in the same space. The predicted rating is then:

$$\hat{r} * ui = \boldsymbol{u}u \cdot \boldsymbol{v}i = \sum f = 1^k uuf\, v * if \tag{9.7}$$

Figure 9.4 illustrates the decomposition.

Figure 9.4. Matrix factorization decomposes the $n \times m$ rating matrix $\mathbf{R}$ into a user factor matrix $\mathbf{U}$ ($n \times k$) and an item factor matrix $\mathbf{V}^\top$ ($k \times m$). k is typically much smaller than n or m.

Learning the Factorization

Only the *observed* ratings $\Omega = \{(u, i) : r_{ui} \text{ is known}\}$ are used during training. The standard objective is the regularised mean squared error:

$$J(\mathbf{U}, \mathbf{V}) = \frac{1}{|\Omega|} \sum_{(u,i)\in\Omega} \big(r_{ui} - \boldsymbol{u}_u \cdot \boldsymbol{v}_i\big)^2 - \lambda\left(\sum_u \|\boldsymbol{u}_u\|^2 + \sum_i \|\boldsymbol{v}_i\|^2\right) \tag{9.8}$$

The ℓ_2 regularisation term controlled by $\lambda > 0$ prevents overfitting by penalising large factor values.

Gradient descent updates for a single observed rating (u, i) are:

$$e_{ui} = r_{ui} - \boldsymbol{u}_u \cdot \boldsymbol{v}_i \tag{9.9}$$

$$\boldsymbol{u}_u \leftarrow \boldsymbol{u} * u + \alpha\big(e * ui\, \boldsymbol{v}_i - \lambda\, \boldsymbol{u}_u\big) \tag{9.10}$$

$$\boldsymbol{v}_i \leftarrow \boldsymbol{v} * i + \alpha\big(e * ui\, \boldsymbol{u}_u - \lambda\, \boldsymbol{v}_i\big) \tag{9.11}$$

where α is the learning rate. These steps are repeated over all observed ratings until convergence (stochastic gradient descent) or in batches (mini-batch gradient descent).

Incorporating Bias Terms

Real rating data contains systematic biases: some users rate everything generously; some items are globally popular and receive uniformly higher scores. An augmented model accounts for these effects:

$$\hat{r}_{ui} = \mu + b_u + b_i + \boldsymbol{u}_u \cdot \boldsymbol{v}_i \tag{9.12}$$

where μ is the global mean rating, b_u is the user bias, and b_i is the item bias. All bias terms are jointly learned with $\mathbf{U}$ and $\mathbf{V}$.

Alternating Least Squares

An alternative optimisation strategy is *Alternating Least Squares* (ALS): hold $\mathbf{V}$ fixed and solve for $\mathbf{U}$ in closed form, then swap and solve for $\mathbf{V}$, alternating until convergence. Each sub-problem is a standard ridge regression. ALS is particularly well-suited to parallel computation

and is widely used in large-scale systems (e.g., Apache Spark's MLlib). With $\mathbf{V}$ fixed, the optimal user vector for user u is:

$$\boldsymbol{u} * u = \left(\mathbf{V} * {\Omega_u}^\top \mathbf{V} * \Omega_u + \lambda \mathbf{I}\right)^{-1} \mathbf{V} * {\Omega_u}^\top \boldsymbol{r}_u^{\Omega_u} \tag{9.13}$$

where $\mathbf{V}_{\Omega_u}$ contains only the rows of $\mathbf{V}$ corresponding to items rated by user u, and $\boldsymbol{r}_u^{\Omega_u}$ is the vector of those observed ratings.

9.1.4 Binary and Implicit Feedback

So far we have assumed *explicit* ratings (e.g., 1–5 stars). In many practical applications only *implicit* feedback is available: click streams, purchase histories, watch times. Here the absence of an observation does not mean the user dislikes an item – it may simply mean the user has not yet discovered it. A common model for implicit feedback treats observed interactions as positive evidence and unobserved pairs as uncertain negatives. The confidence matrix $\mathbf{C}$ is defined as:

$$c_{ui} = 1 + \alpha_{\text{conf}}\, d_{ui} \tag{9.14}$$

where d_{ui} is a measure of interaction (e.g., number of plays), and α_{conf} is a hyperparameter. The binary preference indicator is $p_{ui} = \mathbf{1}[d_{ui} > 0]$. The weighted least squares objective becomes:

$$J = \sum_{u,i} c_{ui}\,(p_{ui} - \boldsymbol{u}_u \cdot \boldsymbol{v}_i)^2 + \lambda\left(\sum_u \|\boldsymbol{u}_u\|^2 + \sum_i \|\boldsymbol{v}_i\|^2\right) \tag{9.15}$$

9.2 Recommender Systems: Implementation Details

9.2.1 Problem Formulation and Notation

Throughout this section we adopt the following notation. Let there be n users and m movies. We introduce:

- $r^{(i,j)} \in \{0,1\}$: indicator that user j has rated movie i.
- $y^{(i,j)}$: the rating given by user j to movie i (defined only when $r^{(i,j)} = 1$).
- $\boldsymbol{w}^{(j)} \in \mathbb{R}^k$ and $b^{(j)} \in \mathbb{R}$: learnable parameter vector and bias for user j.
- $\boldsymbol{x}^{(i)} \in \mathbb{R}^k$: feature vector for movie i.
- The predicted rating is $\hat{y}^{(i,j)} = \boldsymbol{w}^{(j)} \cdot \boldsymbol{x}^{(i)} + b^{(j)}$.

9.2.2 Cost Function

The cost function is minimised over all *observed* user–movie pairs:

$$J\Big(\{w^{(j)}, b^{(j)}\} * j = 1^{n_u}, \{x^{(i)}\} * i = 1^{n_m}\Big) = \frac{1}{2} \sum_{(i,j):\, r^{(i,j)}=1} \Big(\boldsymbol{w}^{(j)} \cdot \boldsymbol{x}^{(i)} + b^{(j)} - y^{(i,j)}\Big)^2 - \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k} (w_k^{(j)})^2 - \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k} \tag{9.16}$$

The first term measures prediction error on known ratings. The second and third terms are ℓ_2 regularisation on user parameters and item features, respectively. Crucially, unlike in supervised learning, *both* the user parameters $\{\boldsymbol{w}^{(j)}, b^{(j)}\}$ and the item features $\{x^{(i)}\}$ are simultaneously optimised.

Remark 9.2.1. When item features $\boldsymbol{x}^{(i)}$ are *fixed* (known from content descriptors), minimising over only $\{\boldsymbol{w}^{(j)}, b^{(j)}\}$ yields a *content-based* linear regression model. When both are learned, the resulting algorithm is a form of *collaborative filtering* via matrix factorization.

9.2.3 Mean Normalisation

Sparse ratings cause a practical difficulty: if a user has rated no items, their parameters $\boldsymbol{w}^{(j)}$ and $b^{(j)}$ are determined solely by regularisation and default to zero, giving the unhelpful prediction $\hat{y}^{(i,j)} = 0$ for all items. *Mean normalisation* resolves this by centring each movie's ratings before learning:

$$\mu_i = \frac{\sum_{j:\, r^{(i,j)}=1} y^{(i,j)}}{\sum_j r^{(i,j)}} \tag{9.17}$$

$$y^{(i,j)}_{\text{norm}} = y^{(i,j)} - \mu_i \tag{9.18}$$

The model is trained on $y^{(i,j)}_{\text{norm}}$, and predictions are obtained by adding the mean back:

$$\hat{y}^{(i,j)} = \boldsymbol{w}^{(j)} \cdot \boldsymbol{x}^{(i)} + b^{(j)} + \mu_i \tag{9.19}$$

Now a new user with no ratings receives the prediction $\hat{y}^{(i,j)} = \mu_i$, i.e., the average rating for movie i — a sensible default.

9.2.4 Gradient Descent Implementation

All parameters $\boldsymbol{w}^{(j)}$, $b^{(j)}$, and $\boldsymbol{x}^{(i)}$ are initialised randomly (small random values break symmetry) and updated jointly via gradient descent. The gradients of the cost with respect to each

set of parameters are:

$$\frac{\partial J}{\partial w_k^{(j)}} = \sum_{i:\, r^{(i,j)}=1} \left(\hat{y}^{(i,j)} - y^{(i,j)}\right) x_k^{(i)} + \lambda\, w_k^{(j)} \tag{9.20}$$

$$\frac{\partial J}{\partial b^{(j)}} = \sum_{i:\, r^{(i,j)}=1} \left(\hat{y}^{(i,j)} - y^{(i,j)}\right) \tag{9.21}$$

$$\frac{\partial J}{\partial x_k^{(i)}} = \sum_{j:\, r^{(i,j)}=1} \left(\hat{y}^{(i,j)} - y^{(i,j)}\right) w_k^{(j)} + \lambda\, x_k^{(i)} \tag{9.22}$$

In TensorFlow or PyTorch, these derivatives are computed automatically via automatic differentiation, eliminating the need for hand-coded gradients.

Example 9.2.1 (TensorFlow Implementation Sketch)**.** The following pseudocode outlines a TensorFlow/Keras implementation.

Initialise $\mathbf{W} \in \mathbb{R}^{n_u \times k}$, $\boldsymbol{b} \in \mathbb{R}^{n_u}$, $\mathbf{X} \in \mathbb{R}^{n_m \times k}$ randomly.
for each iteration **do**
    with GradientTape():
        Compute $\hat{\mathbf{Y}} = \mathbf{X}\mathbf{W}^\top + \boldsymbol{b}$ (shape: $n_m \times n_u$)
        Compute cost J using only rated (i, j) pairs
    end with
    Retrieve gradients $\nabla_{\mathbf{W}} J$, $\nabla_{\boldsymbol{b}} J$, $\nabla_{\mathbf{X}} J$
    $\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} J$
    $\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \nabla_{\boldsymbol{b}} J$
    $\mathbf{X} \leftarrow \mathbf{X} - \alpha \nabla_{\mathbf{X}} J$
end for

9.2.5 Finding Related Items

A useful by-product of the learned item feature matrix $\mathbf{X}$ is the ability to identify items similar to a query item i. Two items are considered similar if their latent feature vectors are close in Euclidean space:

$$\text{sim}(i, i') = \left\| \boldsymbol{x}^{(i)} - \boldsymbol{x}^{(i')} \right\|_2^2 \tag{9.23}$$

The K items with the smallest distance to item i form its neighbourhood of related items. This is used, for example, in the "customers who liked this also liked ..." feature.

9.2.6 Limitations of Collaborative Filtering

Despite its success, collaborative filtering has well-known limitations:

- **Cold-start problem.** New users or items with few or no ratings cannot be handled well by CF, since no collaborative signal exists.
- **Scalability.** Computing all pairwise similarities among millions of users or items is computationally expensive.

- **Popularity bias.** CF tends to over-recommend popular items and under-recommend niche ones, since popular items have more co-occurrence signal.
- **Filter bubbles.** By relying exclusively on past behaviour, CF may reinforce existing preferences and limit the diversity of recommendations.

Content-based filtering, discussed next, mitigates some of these issues.

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. Rather than relying on what *other* users have done, it asks: *"What are the properties of items this particular user has enjoyed, and which other items share those properties?"* This approach requires a rich description of item content. For movies, this might include genre, director, cast, release year, and plot keywords. For news articles, it might be the bag-of-words (TF-IDF) representation of the text. For products, it might be category, price range, and brand.

9.3.2 Item Feature Representation

Let each item i be described by a feature vector $\boldsymbol{x}^{(i)} \in \mathbb{R}^d$. These features can be:

- **Hand-crafted**: domain experts define relevant attributes.
- **Learned**: neural embeddings from item content (e.g., BERT for text, CNNs for images).

Example 9.3.1 (Movie Feature Vector)**.** Consider a five-dimensional feature representation for a movie:

$$\boldsymbol{x}^{(i)} = [\underbrace{0.9}_{}*\text{action},\ \underbrace{0.1}_{}*\text{romance},\ \underbrace{0.7}_{}*\text{sci-fi},\ \underbrace{0.3}_{}*\text{comedy},\ \underbrace{0.0}_{\text{documentary}}\ ]$$

Each entry represents the degree to which the movie belongs to a given genre, obtained, say, from crowd-sourced annotations.

9.3.3 User Profile Estimation

A user profile $\boldsymbol{\theta}^{(j)} \in \mathbb{R}^d$ is estimated from the user's rating history. The simplest approach is to train a linear regression model for each user:

$$\hat{y}^{(i,j)} = \boldsymbol{\theta}^{(j)} \cdot \boldsymbol{x}^{(i)} + b^{(j)} \tag{9.24}$$

The profile $\boldsymbol{\theta}^{(j)}$ encodes how much user j values each feature. It is estimated by minimising the squared error over all items rated by user j:

$$\min_{\boldsymbol{\theta}^{(j)},\, b^{(j)}} \frac{1}{2} \sum_{i:\, r^{(i,j)}=1} \left(\boldsymbol{\theta}^{(j)} \cdot \boldsymbol{x}^{(i)} + b^{(j)} - y^{(i,j)}\right)^2 - \frac{\lambda}{2} \left\|\boldsymbol{\theta}^{(j)}\right\|^2 \tag{9.25}$$

Once trained, the model predicts the user's rating on any new item from its feature vector alone, *without needing other users' data.*

Figure 9.5 illustrates how item feature vectors cluster in a two-dimensional feature space, and how a user's profile (star) aligns with nearby items.

Figure 9.5. Content-based filtering in a 2D feature space. Each point is a movie; colours denote genre. The black star is the user's inferred profile. The dashed circle indicates the neighbourhood of recommended items.

9.3.4 Deep Content-Based Filtering with Neural Networks

Modern content-based systems use deep neural networks to jointly learn user embeddings and item embeddings from rich, heterogeneous feature inputs (text, images, metadata). The predicted score for a user–item pair is:

$$\hat{y}^{(i,j)} = f_\phi\big(g_\psi(\boldsymbol{x}^{(j)} * \text{user}),\; h * \omega(\boldsymbol{x}^{(i)}_{\text{item}})\big) \tag{9.26}$$

where g_ψ and h_ω are neural networks mapping raw features to embeddings, and f_ϕ is an interaction function (e.g., dot product, concatenation followed by a linear layer).

9.3.5 Advantages and Limitations

Advantages:

- Handles the *cold-start* problem for new items: as long as item features are available, predictions can be made immediately.
- Recommendations are *explainable*: "recommended because you liked action movies."
- Does not suffer from popularity bias; niche items with matching features can be recommended.

Limitations:

- Requires rich, high-quality item features, which may be expensive to obtain or encode.
- Tends to produce *over-specialisation*: the system keeps recommending items similar to what the user has already seen, reducing serendipity.
- The user cold-start problem remains: a new user with no rating history has an uninformative profile $\boldsymbol{\theta}^{(j)}$.

9.3.6 Hybrid Systems

In practice, production systems combine collaborative and content-based filtering to balance their respective strengths. Google's YouTube recommender, for example, uses a two-tower deep neural network where one tower processes user features (including watch history encoded via learned embeddings) and the other processes video content features; the final score is a dot product of the two tower outputs.

9.4 Principal Component Analysis

9.4.1 Motivation

Real-world data, including the high-dimensional user–item interaction matrices of recommender systems, often contain far fewer *independent* sources of variation than the raw number of features suggests. For example, thousands of movies can be described by a handful of latent concepts such as 'action-oriented,"' critically acclaimed," or "family-friendly." This latent structure is low-dimensional, and recovering it enables better generalisation, less noise, and faster computation.

Principal Component Analysis (PCA) is the most widely used linear dimensionality reduction technique. It finds a new orthonormal coordinate system aligned with the directions of maximum variance in the data.

9.4.2 Problem Setup

Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ be a data matrix with n samples and d features. We assume the data has been *mean-centred*: $\sum_{i=1}^{n} x_{ij} = 0$ for each feature j (or equivalently, the sample mean has been subtracted from each column). The empirical covariance matrix is:

$$\boldsymbol{\Sigma} = \frac{1}{n-1} \mathbf{X}^{\top} \mathbf{X} \in \mathbb{R}^{d \times d} \tag{9.27}$$

PCA seeks the directions $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_k \in \mathbb{R}^d$ (called *principal components*) along which the projected data has maximum variance, subject to the constraints that each $\|\boldsymbol{u}_l\| = 1$ and $\boldsymbol{u}{*}l \perp \boldsymbol{u}{*}l'$ for $l \neq l'$.

9.4.3 Derivation via Eigendecomposition

The first principal component $\boldsymbol{u}_1$ solves:

$$\max_{\boldsymbol{u}_1 \in \mathbb{R}^d,\, \|\boldsymbol{u}_1\| = 1} \; \boldsymbol{u}_1^{\top} \boldsymbol{\Sigma}\, \boldsymbol{u}_1 \tag{9.28}$$

Using a Lagrange multiplier, the solution satisfies $\boldsymbol{\Sigma}\, \boldsymbol{u}_1 = \lambda_1\, \boldsymbol{u}_1$, i.e., $\boldsymbol{u}_1$ is the eigenvector of $\boldsymbol{\Sigma}$ corresponding to its largest eigenvalue λ_1. The variance of the projection onto $\boldsymbol{u}_1$ equals λ_1.

More generally, the l-th principal component is the eigenvector of $\mathbf{\Sigma}$ with the l-th largest eigenvalue λ_l. The eigenvalues satisfy $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d \geq 0$.

Proposition 9.4.1 (PCA via Eigendecomposition)**.** *Let $\mathbf{\Sigma} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top$ be the eigendecomposition of the covariance matrix, where $\mathbf{\Lambda} = \operatorname{diag}(\lambda_1, \ldots, \lambda_d)$ and the columns of $\mathbf{U}$ are the corresponding eigenvectors. The first k principal components are the first k columns of $\mathbf{U}$. The projection of $\mathbf{X}$ onto this k-dimensional subspace is:*

$$\mathbf{Z} = \mathbf{X}\mathbf{U}_k \tag{9.29}$$

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

9.4.4 Singular Value Decomposition Approach

An equivalent and numerically more stable formulation uses the *Singular Value Decomposition* (SVD) of the (centred) data matrix:

$$\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^\top \tag{9.30}$$

where $\mathbf{U} \in \mathbb{R}^{n \times n}$ and $\mathbf{V} \in \mathbb{R}^{d \times d}$ are orthogonal matrices, and $\mathbf{S} \in \mathbb{R}^{n \times d}$ is a diagonal matrix with non-negative singular values $s_1 \geq s_2 \geq \cdots \geq s_{\min(n,d)} \geq 0$. The principal components are the columns of $\mathbf{V}$ (the right singular vectors), and the singular values are related to the eigenvalues of $\mathbf{\Sigma}$ by $\lambda_l = s_l^2/(n-1)$.

The SVD approach avoids forming the $d \times d$ covariance matrix explicitly, which is beneficial when $d \gg n$ (e.g., in text applications with many features).

9.4.5 Choosing the Number of Components

The fraction of total variance explained by the top k components is:

$$\mathrm{EVR}(k) = \frac{\sum_{l=1}^{k} \lambda_l}{\sum_{l=1}^{d} \lambda_l} \tag{9.31}$$

A common heuristic is to choose k such that $\mathrm{EVR}(k) \geq 0.95$, meaning the reduced representation retains at least 95% of the total variance.

Alternatively, a *scree plot* visualises λ_l against l. One looks for the *elbow* of the curve: the point where adding further components yields diminishing returns.

Figure 9.6 shows both the projection onto the first principal component (left panel) and a scree plot for a two-dimensional example (right panel).

Figure 9.6. Left: projection of 2D data onto the first principal component (PC1). Grey lines indicate perpendicular projections; red dots are the projected points. Right: scree plot showing the variance explained by each component and the cumulative total.

9.4.6 Step-by-Step PCA Algorithm

Algorithm 3 Principal Component Analysis

Require: Data matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$, target dimension k

Ensure: Reduced data matrix $\mathbf{Z} \in \mathbb{R}^{n \times k}$

1: Compute the column-wise mean: $\bar{\boldsymbol{x}} = \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{x}_i$

2: Centre the data: $\mathbf{X} \leftarrow \mathbf{X} - \mathbf{1}\bar{\boldsymbol{x}}^\top$

3: Compute covariance: $\boldsymbol{\Sigma} = \frac{1}{n-1} \mathbf{X}^\top \mathbf{X}$

4: Compute eigendecomposition: $\boldsymbol{\Sigma} = \mathbf{U} \boldsymbol{\Lambda} \mathbf{U}^\top$ (sort eigenvalues in descending order)

5: Extract top-k eigenvectors: $\mathbf{U} * k = \mathbf{U} * :, 1 : k$

6: Project data: $\mathbf{Z} = \mathbf{X}\, \mathbf{U}_k$

7: **return Z**

Figure 9.7. PCA on a 2D dataset. PC1 (red arrow) points in the direction of maximum variance; PC2 (green arrow) is orthogonal to PC1 and captures the residual variance.

9.4.7 Reconstruction and Approximation Error

Given the reduced representation $\mathbf{Z} = \mathbf{X}\mathbf{U}_k$, an approximate reconstruction of the original data is:

$$\hat{\mathbf{X}} = \mathbf{Z}\, \mathbf{U}_k^\top + \mathbf{1}\bar{\boldsymbol{x}}^\top \tag{9.32}$$

The mean squared reconstruction error is:

$$\text{Error} = \frac{1}{n} \left\| \mathbf{X} - \hat{\mathbf{X}} \right\| * F^2 = \frac{1}{n} \sum * l = k + 1^d s_l^2 = \sum_{l=k+1}^{d} \lambda_l \tag{9.33}$$

where $\|\cdot\|_F$ denotes the Frobenius norm. This equals the sum of the *discarded* eigenvalues, confirming that PCA minimises the reconstruction error among all linear projections of rank k.

9.4.8 PCA in the Context of Recommender Systems

PCA and its non-linear generalisations appear in recommender systems in several ways:

1. **Dimensionality reduction of item features.** High-dimensional content features (e.g., TF-IDF vectors with tens of thousands of dimensions) are compressed with PCA before being fed into a linear regression model.
2. **Latent Semantic Analysis.** Applying truncated SVD to the user–item matrix extracts latent factors analogous to those of matrix factorization. The k-rank approximation directly fills in missing entries:
$$\hat{\mathbf{R}} = \mathbf{U}_k \mathbf{S}_k \mathbf{V}_k^\top \tag{9.34}$$
3. **Visualisation.** Projecting high-dimensional item or user embeddings to 2D or 3D with PCA enables human inspection of the latent space structure.
4. **Noise reduction.** By discarding small singular values, SVD filters out random noise in the rating matrix and can improve generalisation.

9.4.9 Relationship between PCA and Matrix Factorisation

There is a deep connection between PCA and the matrix factorization approach to collaborative filtering. When the cost function in Equation (9.8) is applied to a *complete* (non-sparse) matrix with no regularisation ($\lambda = 0$), the global minimum is exactly the rank-k truncated SVD of $\mathbf{R}$. This means that collaborative filtering via gradient descent generalises PCA to the incomplete, regularised setting.

Proposition 9.4.2. *For a fully observed matrix $\mathbf{R}$, the rank-k matrix factorization without regularisation recovers the principal subspace of $\mathbf{R}$:*

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

where $\mathbf{U}_k$, $\mathbf{S}_k$, $\mathbf{V}_k$ are the top-k components of the SVD of $\mathbf{R}$.

9.4.10 Limitations of PCA

- PCA is a *linear* method. If the data lies on a non-linear manifold, PCA may not find a compact representation. Non-linear methods such as autoencoders or t-SNE may be more appropriate.
- PCA is sensitive to the scale of features. Features with large variance dominate the principal components unless the data is standardised (i.e., divided by the standard deviation) before applying PCA.
- The principal components are not always interpretable: they are linear combinations of all original features.

- PCA does not directly handle missing data, as required in collaborative filtering. Specialised algorithms such as *EM-PCA* or probabilistic PCA are needed for incomplete matrices.

Summary

This chapter has introduced the foundations of recommender systems. The key takeaways are:

1. **Collaborative Filtering** exploits shared preferences among users to fill in missing ratings. Memory-based methods (user-based and item-based CF) rely on similarity metrics such as cosine similarity and Pearson correlation. Model-based methods, particularly *matrix factorization*, learn compact latent representations optimised by gradient descent or ALS, and scale to millions of users and items.
2. **Implementation Details** include careful formulation of the cost function over observed ratings, the application of mean normalisation to handle new users, and the use of automatic differentiation in frameworks like TensorFlow. Learned item embeddings can also be repurposed to compute item–item similarity.
3. **Content-Based Filtering** builds a per-user preference model from item feature vectors. It handles the item cold-start problem and provides explainable recommendations, but requires rich item features and can suffer from over-specialisation.
4. **Principal Component Analysis** extracts the principal axes of variation in the data via eigendecomposition of the covariance matrix or equivalently via SVD. It provides a theoretical underpinning for matrix factorization and is a key tool for dimensionality reduction, noise filtering, and latent semantic analysis in recommender systems.

In practice, state-of-the-art recommender systems combine all three paradigms in large-scale deep learning architectures that jointly learn user embeddings, item embeddings, and contextual features (time, device, session) from billions of interactions.

Further Reading

- Y. Koren, R. Bell, and C. Volinsky, 'Matrix Factorization Techniques for Recommender Systems," *IEEE Computer*, 2009.
- G.
  Adomavicius and A.
  Tuzhilin,' Toward the Next Generation of Recommender Systems: A Survey," *IEEE TKDE*, 2005.
- X. He *et al.*, "Neural Collaborative Filtering," *WWW*, 2017.
- C. Aggarwal, *Recommender Systems: The Textbook*, Springer, 2016.
- I. Jolliffe, *Principal Component Analysis*, 2nd ed., Springer, 2002.

CHAPTER 10

Reinforcement Learning

Chapter Overview. Reinforcement learning trains agents to act in sequential decision environments by maximising cumulative reward. This chapter covers the Markov Decision Process framework, the Bellman equation, Q-learning, and the Deep Q-Network (DQN) with its engineering refinements: experience replay, mini-batch updates, ε-greedy exploration, and soft target updates.

Chapter Overview

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 in unlabelled data, reinforcement learning trains an *agent* to take sequential decisions by interacting with an *environment* and receiving feedback in the form of numerical *rewards*. This chapter builds RL from the ground up: we begin with intuition and the core vocabulary, develop the formal mathematical framework (Markov Decision Processes), derive the Bellman equation, extend the framework to continuous state spaces, and conclude with a complete treatment of Deep Q-Networks and the engineering refinements required to make them work in practice.

10.1 Part I: Reinforcement Learning Introduction

10.1.1 What Is Reinforcement Learning?

Core Philosophy

Reinforcement Learning is concerned with training an autonomous **agent** to make a sequence of decisions inside an **environment** in order to maximise a cumulative numerical signal called the **reward**. The key distinguishing feature is that the system is told *what to achieve* (via the reward function), not *how to achieve it* (via explicit step-by-step instructions or labelled demonstrations).

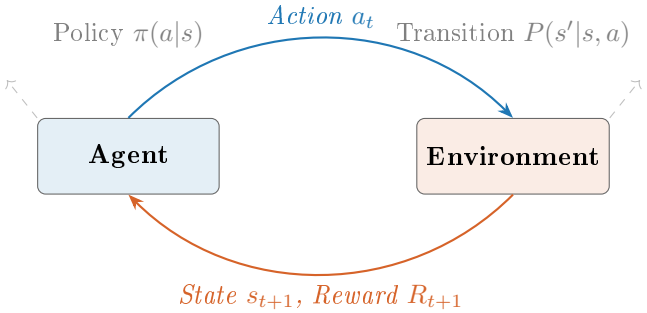


Figure 10.1. The reinforcement learning interaction loop. The **agent** observes the current state s_t, selects an action a_t according to its policy π, and receives from the **environment** a reward R_{t+1} and a new state s_{t+1}. This cycle repeats until a terminal state is reached.

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

Analogy. Training a dog provides an apt mental model. When the dog sits on command it receives a treat (positive reward); when it misbehaves it is scolded (negative reward). Over many repetitions the dog learns which behaviours earn treats, without anyone programming the exact muscle movements required.

Key Components and Mathematical Notation

The RL framework is built from five building blocks.

1. **State** (s or x): A complete description of the agent's current situation. For an autonomous helicopter this includes position, orientation, speed, and acceleration.
2. **Action** (a or y): The decision the agent makes at each time step. For the helicopter, an action corresponds to a particular joystick movement.
3. **Reward** $R(s)$: A scalar feedback signal.
   - A *positive* reward encourages behaviour (e.g. $+1$ per second of stable flight).
   - A *negative* reward penalises failure (e.g. -1000 for a crash).
4. **Policy** π: A function mapping states to actions (defined formally in Section 10.1.3).
5. **Return** G: The (discounted) cumulative reward the agent seeks to maximise (defined formally in Section 10.1.2).

Reinforcement Learning vs. Supervised Learning

Although supervised learning (SL) is widespread, it is unsuitable for many sequential decision problems. Table 10.1 summarises the core differences.

A fundamental limitation of SL for control tasks is the need for a dataset of *ideal* actions. For complex motor tasks (robotic locomotion, aerobatic flight) it is often impossible to specify the optimal action analytically in every state. The reward function sidesteps this: instead of

Table 10.1. Comparison of Supervised Learning and Reinforcement Learning.

Feature	Supervised Learning	Reinforcement Learning
Input mapping	$x \mapsto y$ (label)	$s \mapsto a$ (action)
Data required	Expert-labelled dataset	Reward function + experience
Ambiguity	Difficult to handle	Resolves by exploration
Objective	Predict correct label	Maximise total reward

dictating *how* to fly, we simply reward steady flight and penalise crashes, letting the algorithm discover the control strategy autonomously.

Real-World Applications

RL has produced landmark results across diverse domains:

- **Robotics**: aerobatic helicopter manoeuvres, quadruped locomotion over rough terrain, lunar lander simulations.
- **Industrial optimisation**: factory logistics scheduling to maximise throughput.
- **Finance**: sequencing large equity trades to minimise market impact.
- **Games**: mastering Chess, Go, Bridge, and modern video games beyond human level.

Illustrative Case Study: The Mars Rover

To build intuition we use a simplified one-dimensional Mars Rover throughout this chapter.

- **State space**: six discrete positions, labelled $s \in \{1, 2, 3, 4, 5, 6\}$.
- **Action space**: move LEFT or move RIGHT.
- **Terminal states**: $s = 1$ and $s = 6$ end the episode after their reward is collected.
- **Rewards**: $R(1) = 100$ (high-value science target), $R(6) = 40$ (lower-value target), $R(s) = 0$ for $s \in \{2, 3, 4, 5\}$.

Figure 10.2 illustrates the environment.

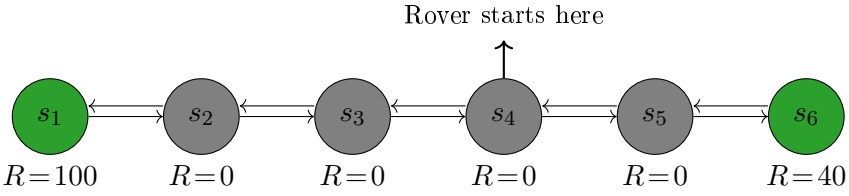


Figure 10.2. The Mars Rover environment. Terminal states s_1 (reward 100) and s_6 (reward 40) are shaded. The rover must choose at each step to move left or right.

A key behavioural insight emerges immediately: if the rover starts at $s = 4$, it can reach either terminal state. A naive agent might go right because s_6 is closer (two steps), ignoring the larger reward at s_1 (three steps). Whether a well-designed algorithm prefers the larger, more distant reward depends critically on the *discount factor*, introduced next.

10.1.2 The Return in Reinforcement Learning

Motivation

An agent that only cares about its immediate reward is dangerously short-sighted. The **return** provides a scalar summary of a complete reward sequence that correctly weights near-term rewards more highly than distant ones.

The Discount Factor γ

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

- γ close to 1 (e.g. 0.99): the agent is *far-sighted*; future rewards are almost as valuable as immediate ones.
- γ close to 0 (e.g. 0.5): the agent is *myopic*; rewards a few steps away are heavily discounted.

Financial analogy. γ mirrors the *time value of money*: a dollar today is worth more than a dollar in a year, because today's dollar can earn interest. Equivalently, a reward obtained now is more valuable than the same reward obtained after k time steps.

Mathematical Definition

Given a trajectory of rewards $R_1, R_2, R_3, \ldots$ until a terminal state, the **return** G is:

$$\boxed{G = R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4 + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{k+1}} \tag{10.1}$$

The reward at step k is multiplied by γ^{k-1}, so later rewards contribute progressively less to the total.

Mars Rover Example

Consider $\gamma = 0.9$. Starting at $s = 4$ and moving left, the reward sequence is $[0, 0, 0, 100]$ (zero reward for states $3, 2$ and terminal reward 100 at s_1):

$$G = 0 + 0.9 \cdot 0 + 0.9^2 \cdot 0 + 0.9^3 \cdot 100 = 72.9.$$

Table 10.2 compares returns for different policies with $\gamma = 0.5$.

Table 10.2. Returns from different starting states and policies ($\gamma = 0.5$).

Start state	Policy	Reward sequence	Return G
$s = 4$	Always Left	$[0, 0, 0, 100]$	$0.5^3 \times 100 = 12.5$
$s = 2$	Always Left	$[0, 100]$	$0.5^1 \times 100 = 50.0$
$s = 4$	Always Right	$[0, 0, 40]$	$0.5^2 \times 40 = 10.0$
$s = 5$	Always Right	$[0, 40]$	$0.5^1 \times 40 = 20.0$

Handling Negative Rewards

When the reward function contains negative values (penalties), the discount factor creates an incentive to *postpone* negative events. Because $\gamma < 1$, a penalty of -10 incurred in ten steps contributes approximately $\gamma^{10} \times (-10)$ to the return—a much smaller magnitude than -10 today. RL agents naturally learn to delay unavoidable penalties, which is often the desired behaviour.

10.1.3 Making Decisions: Policies in Reinforcement Learning

Definition of a Policy

A **policy** π is a function that maps every state to the action the agent should take in that state:

$$\pi(s) = a.$$

The policy fully specifies the agent's behaviour: given any state s the agent simply executes action $\pi(s)$, with no need to look at the history.

Example Policies for the Mars Rover

- $\pi(s) = \leftarrow$ for all s: always go left.
- $\pi(2) = \leftarrow,\ \pi(3) = \leftarrow,\ \pi(4) = \leftarrow,\ \pi(5) = \rightarrow$: go left from states 2–4, right from state 5.

Common Policy Heuristics

1. *Proximity-based*: always move toward the nearest reward.
2. *Magnitude-based*: always move toward the largest reward, ignoring distance.
3. *Directional*: always move in one fixed direction.
4. *Conditional*: a default direction with exceptions near specific states.

None of these heuristics is guaranteed to be optimal; the goal of RL is to discover the *best* policy automatically.

The Goal of Reinforcement Learning

> *Find the optimal policy π^* that, for every state s, selects the action $a = \pi^*(s)$ that maximises the expected return.*

10.1.4 Review of Key Concepts

Markov Decision Processes (MDPs)

The standard mathematical formalism for RL problems is the **Markov Decision Process (MDP)**.

- **The Markov property**: the future depends only on the present state, not on the history of how the agent arrived there. Decisions are made solely on s_t.
- **The agent–environment loop**: at each time step the agent observes s, selects $a = \pi(s)$, the environment transitions to s', and the agent receives reward $R(s)$. This cycle repeats until a terminal state is reached.

Figure 10.3 shows the agent–environment interaction.

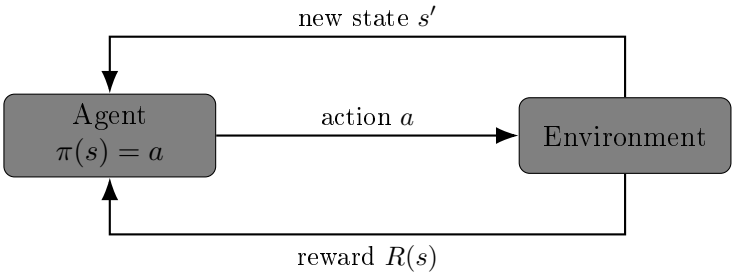


Figure 10.3. The agent–environment loop in a Markov Decision Process.

MDP Components Across Domains

Table 10.3 shows how the MDP components map to several classic problems.

Table 10.3. MDP components in three application domains.

Component	Mars Rover	Helicopter	Chess
States s	6 positions	Position, orientation, speed	Board configuration
Actions a	Left / Right	Joystick inputs	Legal moves
Rewards R	$100, 0, 40$	$+1/-1000$	$+1/0/-1$
γ	0.5	0.99	0.99–0.999

10.2 Part II: State-Action Value Function

10.2.1 State-Action Value Function Definition

From Policy to Value

Knowing the return of a policy allows us to compare policies, but we need a structured way to do so for every state and every action. The **State-Action Value Function** (also called the **Q-function**) provides exactly this.

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

1. Starting in state s.
2. Taking action a exactly once.
3. Following the *optimal policy* for all subsequent steps.

Deriving the Optimal Policy from Q

Once Q is known, optimal action selection is trivial:

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

$$\text{Max return from } s = \max_a Q(s, a). \tag{10.3}$$

The agent simply evaluates $Q(s, a)$ for all possible actions a and picks the one with the highest value. The Q-function is also referred to in the literature as the **optimal Q-function** $Q^*(s, a)$.

10.2.2 State-Action Value Function Example

Table 10.4 lists Q-values for several state-action pairs in the Mars Rover with $\gamma = 0.5$, terminal rewards $R(1) = 100$ and $R(6) = 40$.

Table 10.4. Selected Q-values for the Mars Rover ($\gamma = 0.5$).

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

Observation. Even when the immediate reward is zero (states 2–5), the Q-function is positive because it accounts for the ability to reach a high-reward terminal state optimally in the future. The agent at $s = 2$ correctly prefers to go left ($Q = 50$) over right ($Q = 12.5$).

10.2.3 The Bellman Equation

Motivation

Computing Q-values by exhaustively simulating every possible trajectory is intractable in large environments. The **Bellman Equation** provides a *recursive* decomposition that expresses the Q-value of (s, a) in terms of the Q-values of the next state.

Derivation

Recall the return definition (Equation 10.1):

$$G = R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots$$

Factor out γ from all terms after the first:

$$G = R_1 + \gamma\big(R_2 + \gamma R_3 + \gamma^2 R_4 + \cdots\big).$$

The term in parentheses is the optimal return starting from the next state s':

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

This gives the **Bellman Equation**:

$$\boxed{Q(s, a) = R(s) + \gamma \max_{a'} Q(s', a')} \tag{10.4}$$

Terminal states. When s is terminal there is no successor, so:

$$Q(s, a) = R(s).$$

Component Breakdown

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

γ Discount factor; how much the agent values the future.

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

Worked Example

Compute $Q(4, \leftarrow)$ for the Mars Rover with $\gamma = 0.5$.

$$
\begin{aligned}
Q(4, \leftarrow) &= R(4) + 0.5 \times \max_{a'} Q(3, a') \\
&= 0 + 0.5 \times \max[Q(3, \leftarrow),\, Q(3, \rightarrow)] \\
&= 0 + 0.5 \times 25 = 12.5.
\end{aligned}
$$

The calculation confirms the value in Table 10.4: even though the immediate reward is zero, the Q-value is positive because the agent can still reach the high-value terminal state s_1 optimally.

Python Implementation (Value Iteration)

The following code implements a tabular Bellman update for the Mars Rover.

```
import numpy as np

# Environment

num_states  = 6
num_actions = 2            # 0: Left, 1: Right
gamma       = 0.5
rewards     = np.array([100, 0, 0, 0, 0, 40])

# Q-table initialised to zero

q_table = np.zeros((num_states, num_actions))

def get_next_state(s, a):
if a == 0:    # Left
return max(0, s - 1)
else:         # Right
return min(num_states - 1, s + 1)

def update_q(s, a):
s_next = get_next_state(s, a)
is_terminal = (s_next == 0 or s_next == 5)
r = rewards[s_next] if is_terminal else 0
future = 0 if is_terminal else np.max(q_table[s_next])
q_table[s, a] = r + gamma * future

# Value iteration

for _ in range(20):
for s in range(1, num_states - 1):
for a in range(num_actions):
update_q(s, a)

# Derive optimal policy

policy = {s: np.argmax(q_table[s]) for s in range(1, num_states - 1)}
print("Optimal policy (0=Left, 1=Right):", policy)
```

Listing 10.1. Value iteration for the Mars Rover.

Tabular vs. Deep Q-Learning

Table 10.5. Comparison of tabular Q-learning and Deep Q-Networks.

Feature	Tabular Q-Learning	Deep Q-Network (DQN)
State space	Small, discrete	Large or continuous
Storage	2-D array	Neural network
Lookup	`q_table[s, a]`	`model.predict(s)`
Scalability	Poor	Excellent

10.2.4 Stochastic (Random) Environments

Definition and Dynamics

In a **stochastic environment** the outcome of an action is not deterministic. Taking action a in state s produces next state s' only with some probability; other transitions may occur due to external factors such as wind, sensor noise, or mechanical slippage.

Example. In state $s = 4$, action LEFT:

- 90% chance of reaching state $s' = 3$ (intended).
- 10% chance of reaching state $s' = 5$ (slip to the right).

Expected Return

Because the reward sequence is now a random variable, we cannot simply maximise a single fixed return. Instead we maximise the **expected return**:

$$\mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{k+1}\right].$$

Stochastic Bellman Equation

The Bellman equation is modified to include the expectation over all possible next states:

$$\boxed{Q(s,a) = R(s) + \gamma\, \mathbb{E}\left[\max_{a'} Q(s', a')\right]} \tag{10.5}$$

The expectation is taken over the transition distribution $P(s' \mid s, a)$.

Impact of Randomness

Table 10.6. Comparison: deterministic vs. stochastic environments.

Feature	Deterministic	Stochastic
Outcome of a	Fixed s'	Distribution over s'
Return	Single value	Random variable
Bellman eq.	$R(s) + \gamma \max Q(s', a')$	$R(s) + \gamma \mathbb{E}[\max Q(s', a')]$
Agent control	Full	Partial

As the probability of unintended transitions increases, Q-values generally *decrease* because the agent has less reliable access to high-reward states.

10.3 Part III: Continuous State Spaces

10.3.1 Examples of Continuous State Space Applications

Discrete vs. Continuous

A **discrete state space** contains a finite, countable number of possible states (e.g. the six grid positions of the Mars Rover). A **continuous state space** allows the state to take any real value within a range, requiring the state to be represented as a *vector* of real numbers.

Self-Driving Truck (2D)

A planar vehicle requires six state variables:

$$s = \begin{bmatrix} x \\ y \\ \theta \\ \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix}$$

where (x, y) is the position, θ is the heading, and $(\dot{x}, \dot{y}, \dot{\theta})$ are the corresponding velocities. The dot notation $\dot{z} \equiv dz/dt$ is standard in physics and control theory.

Autonomous Helicopter (3D)

A helicopter operating in three-dimensional space needs twelve state variables:

$$s = \left[x,\ y,\ z,\ \phi,\ \theta,\ \omega,\ \dot{x},\ \dot{y},\ \dot{z},\ \dot{\phi},\ \dot{\theta},\ \dot{\omega}\right]^\top$$

- *Position*: x (North–South), y (East–West), z (altitude).
- *Attitude*: ϕ (roll), θ (pitch), ω (yaw).
- *Velocities*: linear $(\dot{x}, \dot{y}, \dot{z})$ and angular $(\dot{\phi}, \dot{\theta}, \dot{\omega})$.

Continuous State MDPs

In a continuous state MDP the policy remains $\pi(s) = a$, but now s is a real-valued vector of potentially high dimension. A lookup table is no longer feasible; a **function approximator** (typically a neural network) must be used to represent $Q(s, a)$ over the continuous input space.

10.3.2 Lunar Lander: A Complete Case Study

Environment Overview

The Lunar Lander is a classic RL simulation in which an agent controls a spacecraft and must land safely on a designated pad between two flags. The problem is modelled as a continuous-state MDP.

State Space

The state is an 8-dimensional vector observed at each time step t:

$$s = \left[x,\ y,\ \dot{x},\ \dot{y},\ \theta,\ \dot{\theta},\ l,\ r\right]$$

- (x, y): horizontal and vertical position of the lander.
- $(\dot{x}, \dot{y})$: horizontal and vertical velocity.
- θ: tilt angle (orientation).
- $\dot{\theta}$: angular velocity.
- $l, r \in \{0, 1\}$: binary contact indicators for the left and right landing legs.

Action Space

At each time step the agent chooses one of four discrete actions:

1. **Do nothing**: no engines fire.
2. **Fire left thruster**: pushes the lander to the right.
3. **Fire main engine**: pushes the lander upward.
4. **Fire right thruster**: pushes the lander to the left.

Reward Function

The reward function is engineered to incentivise safe, efficient landing:

Table 10.7. Reward structure for the Lunar Lander.

Event	Reward	Purpose
Reaching landing pad	+100 to +140	Primary objective
Moving toward pad	Positive	Encourage progress
Moving away from pad	Negative	Discourage drifting
Crash	−100	Severe failure penalty
Soft landing	+100	Bonus for control
Each leg grounded	+10	Encourage stability
Main engine fire	−0.3	Penalise fuel waste
Side thruster fire	−0.03	Minor fuel penalty

Learning Objective

The agent must find policy π^* that maximises the discounted return:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \qquad \gamma = 0.985.$$

The high discount factor (γ close to 1) is appropriate because the landing task requires long-term planning: decisions made early (e.g. the initial burn to slow descent) heavily influence outcomes many steps later.

10.3.3 Learning the State-Value Function

Architecture: Input the State, Output Q-values

A neural network is trained to approximate $Q(s, a)$.

- **Input layer** (x): a 12-dimensional vector formed by concatenating the 8-variable state s and a one-hot encoded action a (4 variables). For example, "fire left thruster" is encoded as $[0, 1, 0, 0]$.
- **Hidden layers**: two fully connected layers, each with 64 units and ReLU activations.
- **Output layer**: a single scalar unit estimating $Q(s, a)$.
- **Action selection**:

$$a = \arg\max_a Q(s, a).$$

Training Target from the Bellman Equation

The Bellman equation provides a natural regression target. We treat the RL update as a supervised learning problem:

Input x	State-action pair (s, a)
Target y	$R(s) + \gamma \max_{a'} Q(s', a')$

Training minimises the mean squared error between the network's prediction $Q(s, a)$ and the target y.

The DQN Algorithm

1. **Initialise** the neural network Q with random weights.
2. **Collect experience**: take actions (initially random) and store tuples $(s, a, R(s), s')$ in a **replay buffer** (capacity $\approx$ 10,000 tuples).
3. **Sample a mini-batch** from the replay buffer.
4. **Compute targets** $y_i = R(s_i) + \gamma \max_{a'} Q(s'_i, a')$ for each tuple.
5. **Train** a new network Q_{new} on the mini-batch using MSE loss.
6. **Update**: set $Q \leftarrow Q_{\text{new}}$ (or use a soft update; see Section 10.3.6).
7. Repeat from step 2.

As the Q-function improves, the estimated targets become more accurate, leading to better policies, which in turn generate better training data. This positive feedback loop eventually converges to the optimal Q-function.

10.3.4 Algorithm Refinement: Improved Neural Network Architecture

Limitation of the Naive Architecture

The architecture described above (12-dimensional input, scalar output) requires *four separate forward passes*—one per action—to identify the best action. This is wasteful, especially during Bellman target computation, where the term $\max_{a'} Q(s', a')$ requires evaluating all actions for every training example.

Optimised Multi-Output Architecture

The standard DQN architecture feeds *only the state* s (8 dimensions) into the network and outputs **all four Q-values simultaneously**:

Advantages

1. **Inference efficiency**: one forward pass suffices to identify $\arg\max_a Q(s, a)$, replacing four passes.
2. **Faster Bellman updates**: computing $\max_{a'} Q(s', a')$ during training requires only one forward pass per example, dramatically reducing training time.

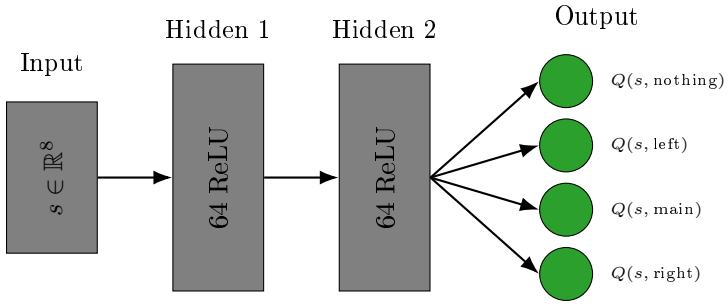


Figure 10.4. Optimised DQN architecture: a single forward pass produces Q-values for all four actions simultaneously.

Table 10.8. Naive vs. optimised DQN architectures.

Feature	Naive	Optimised (Standard)
Input vector	(s, a) concatenated	State s only
Input dimension	$8 + 4 = 12$	8
Output units	1 (scalar)	4 (one per action)
Passes per step	4	1
Main benefit	Conceptual simplicity	Computational efficiency

10.3.5 Algorithm Refinement: ε-Greedy Policy

The Exploration–Exploitation Trade-off

An agent must balance two competing objectives:

- **Exploitation**: choose the action with the currently highest Q-value to maximise immediate performance.
- **Exploration**: try novel actions to discover potentially better strategies.

The Danger of Pure Exploitation

If the network is initialised with random weights, $Q(s, \text{main thruster})$ may happen to be very low by chance. A purely greedy agent will then *never* fire the main thruster and will never discover that it is essential for a soft landing. The agent becomes stuck in a local optimum.

The ε-Greedy Mechanism

With probability ε, take a uniformly random action; with probability $1 - \varepsilon$, take the greedy action:

$$a = \begin{cases} \text{random action} & \text{with probability } \varepsilon, \\ \arg\max_a Q(s, a) & \text{with probability } 1 - \varepsilon. \end{cases}$$

Epsilon Decay Schedule

A common and effective strategy is to *decay* ε over training:

1. **Initial phase**: $\varepsilon = 1.0$. The agent acts almost entirely randomly to build a diverse replay buffer.
2. **Training phase**: gradually reduce ε (e.g. linearly or exponentially toward 0.01).
3. **Final phase**: $\varepsilon = 0.01$. The agent exploits its learned Q-function 99% of the time and explores just 1%.

Figure 10.5 shows a typical exponential decay schedule.

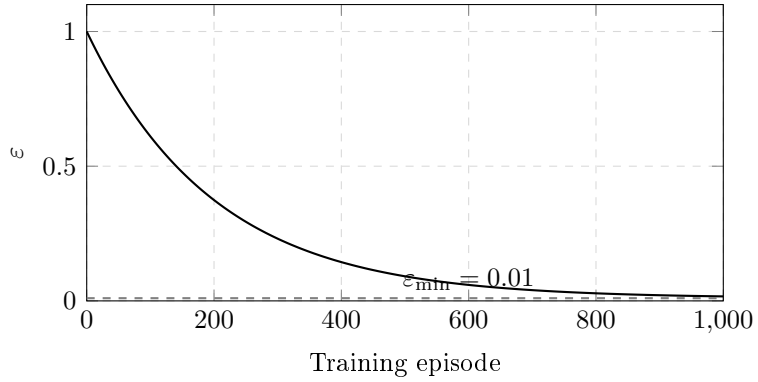


Figure 10.5. Exponential decay of ε from 1.0 toward 0.01 over training.

Table 10.9. Summary of action-selection strategies.

Policy	Logic	Result
Pure greedy	$\arg\max_a Q(s,a)$ always	Risk of local optima
Random	Uniform random	No learning from experience
ε-greedy	Mix of greedy and random	Optimal balance

10.3.6 Algorithm Refinement: Mini-Batch and Soft Updates

Mini-Batch Gradient Descent

The problem.. Standard batch gradient descent computes the loss over the *entire* replay buffer at each step. With a buffer of 10,000 examples (or millions in large-scale applications), each gradient step is extremely expensive.

The solution: mini-batches.. Instead of using all m examples, sample a random mini-batch of size m' (e.g. $m' = 1{,}000$) and compute:

$$J(w,b) = \frac{1}{2m'} \sum_{i=1}^{m'} \big(Q_w(s^{(i)}, a^{(i)}) - y^{(i)}\big)^2.$$

Each gradient step is much cheaper, and many more steps can be taken in the same wall time. Although the loss trajectory is noisier than full-batch descent (Figure 10.6), the algorithm converges faster in practice.

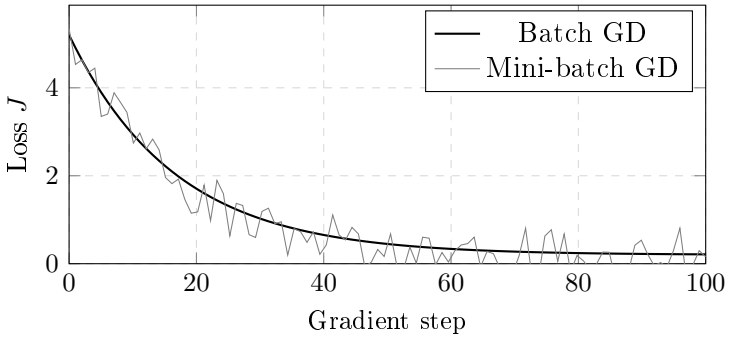


Figure 10.6. Batch gradient descent (smooth) vs. mini-batch gradient descent (noisy). Despite noise, mini-batch converges faster per unit time.

Soft Updates

The problem.. In the original DQN update $Q \leftarrow Q_{\text{new}}$, a single poor mini-batch can suddenly overwrite the entire policy with worse parameters, causing oscillation or divergence.

The solution: soft updates.. Instead of a hard replacement, blend the new parameters into the existing ones with a small step size τ (e.g. $\tau = 0.01$):

$$W \leftarrow \tau\, W_{\text{new}} + (1-\tau)\, W, \tag{10.6}$$

$$b \leftarrow \tau\, b_{\text{new}} + (1-\tau)\, b. \tag{10.7}$$

Only 1% of the new parameters is incorporated at each step; the remaining 99% comes from the stable existing network. This acts as a low-pass filter over the noisy sequence of parameter updates, suppressing instabilities.

Summary of Engineering Refinements

Table 10.10. Key algorithmic refinements in modern DQN.

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

10.4 The State of Reinforcement Learning

10.4.1 Current Application Landscape

Despite dramatic theoretical progress and high-profile successes (AlphaGo, AlphaStar, OpenAI Five), the deployment of RL in industry remains narrower than that of supervised and unsupervised learning. In modern ML pipelines, the overwhelming majority of production systems use supervised or unsupervised learning. For most practical engineering problems, these approaches remain more reliable and easier to tune.

10.4.2 The Simulation-to-Reality Gap

A recurring challenge is that RL algorithms are typically developed and validated in *simulated* environments:

- **Simulation bias**: physics simulators provide unlimited data and ideal conditions. Policies trained there may rely on unrealistic assumptions.
- **Real-world friction**: physical robots experience sensor noise, actuator delays, wear, and environmental variability absent from simulators.
- **Generalisation failure**: a policy achieving 99% success in simulation may perform significantly worse on the real hardware due to the *sim-to-real gap*.

Active research directions addressing this gap include domain randomisation (training across a distribution of simulated environments), system identification (calibrating the simulator to the real system), and sim-to-real transfer techniques.

10.4.3 Future Directions and Potential

RL remains one of the most intellectually exciting frontiers in AI, with unique strengths that complement supervised learning:

- **Autonomous control**: it is the primary framework for problems where an agent must interact with a physical or digital environment over time.
- **Complex sequential decisions**: tasks where immediate consequences are minor but long-term outcomes are critical are natural fits for RL.
- **Robotics**: advanced locomotion, manipulation, and navigation remain core open problems where RL is making rapid progress.
- **Alignment research**: RLHF (Reinforcement Learning from Human Feedback) has become central to training large language models to follow human intent.

10.4.4 Checklist for Applying RL

When deciding whether RL is the right tool, the following questions are a useful guide:

1. **Is a simulator available?** RL is significantly more tractable when a high-fidelity, fast simulator exists.
2. **Is the problem sequential?** Use RL when current actions materially influence future states over many time steps.
3. **Is the reward codifiable?** Success depends on whether the desired behaviour can be expressed as a mathematical scalar reward function.
4. **Is supervised learning insufficient?** If expert demonstrations can cover the relevant state space, imitation learning (a form of SL) may be simpler.

Chapter Summary

This chapter introduced Reinforcement Learning from first principles to practical implementation. The key ideas are:

1. The RL framework consists of **states**, **actions**, **rewards**, a **discount factor** γ, and a **policy** π.
2. The **return** $G = \sum_k \gamma^k R_{k+1}$ is the discounted cumulative reward the agent seeks to maximise.
3. A **Markov Decision Process** (MDP) formalises the agent–environment interaction under the Markov property.
4. The **Q-function** $Q(s, a)$ quantifies the return for taking action a in state s and then acting optimally. The optimal policy satisfies $\pi^*(s) = \arg\max_a Q(s, a)$.
5. The **Bellman equation** $Q(s, a) = R(s) + \gamma \max_{a'} Q(s', a')$ provides a recursive structure that is the foundation of all Q-learning algorithms.
6. In **stochastic environments** the Bellman equation includes an expectation: $Q(s, a) = R(s) + \gamma \mathbb{E}[\max_{a'} Q(s', a')]$.
7. **Continuous state spaces** require function approximation; the **Deep Q-Network (DQN)** uses a neural network to represent Q.
8. Three key engineering refinements make DQN work: the **replay buffer**, **mini-batch updates**, and **soft target updates**.
9. The **ε-greedy policy** balances exploration and exploitation, with ε decayed over training.
10. While RL has enormous potential, its practical deployment is currently narrower than supervised learning, largely due to the **simulation-to-reality gap**.