Notes for Understanding
Machine Learning

A personal learning textbook

Ioanna Stamou

Contents

Introduction

Imagine you have a robot friend.

You give the robot a task:

"Look at these examples, learn the pattern, and make predictions."

This is machine learning.

Machine learning = finding patterns in data.

Example:

You give a model 1000 houses with their *price*, *size*, and *number of rooms*. The model learns the relationship between these features and the price. Then, when you give it a *new* house, it predicts the price.

That's all.

> Key Idea
>
> Machine learning is not magic. It is a systematic way to:
>
> 1. collect data,
> 2. find patterns,
> 3. use those patterns to make predictions or decisions.

The rest of these notes are about:

- the **statistics** you need to talk precisely about patterns,
- the **Python tools** you will use to work with data,
- the **machine learning algorithms** that actually learn from examples.

1

Statistics You Need for Machine Learning

In this chapter we will develop the basic statistical ideas that appear everywhere in machine learning: random variables, distributions, expectation, variance, covariance, correlation, and a few key theorems that explain why learning from data can work.

1.1 Types of Variables

Numerical Variables

Numerical variables take quantitative values.

- **Continuous:** $x \in \mathbb{R}$ Examples: height, temperature, house price.
- **Discrete:** $x \in \mathbb{Z}$ Examples: number of rooms, number of customers.

Categorical Variables

Categorical variables represent non-numerical classes.

- **Nominal (unordered)** Examples: color, country, type of food.
- **Ordinal (ordered)** Examples: rating levels, education level.

Important: Machine learning models expect numerical inputs. Categorical variables must be encoded (one-hot, label encoding, etc.).

1.2 Descriptive Statistics

1.2.1 Mean

The mean (average) of n observations is:

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i. \tag{1.1}$$

1.2.2 Median

The median is the middle value of the sorted data. It is more robust to outliers than the mean.

1.2.3 Variance

Variance measures how far the values are spread from the mean:

$$\mathrm{Var}(X) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2. \tag{1.2}$$

1.2.4 Standard Deviation

Standard deviation is the square root of the variance:

$$\sigma = \sqrt{\mathrm{Var}(X)}. \tag{1.3}$$

It measures the typical deviation from the mean.

1.2.5 Covariance

Variance tells us how a single variable varies. Covariance tells us how *two* variables vary *together*.

$$\mathrm{Cov}(X,Y) = \mathbb{E}\left[(X - \mu_X)(Y - \mu_Y)\right]. \tag{1.4}$$

Interpretation:

- Positive covariance: when X is above its mean, Y tends to be above its mean.
- Negative covariance: when X is above its mean, Y tends to be below its mean.
- Zero covariance: no linear relationship.

1.2.6 Correlation

Correlation measures the strength of linear dependence between two variables.

- Range: $[-1,1]$
- Indicates how two variables move together

$$\rho_{XY} = \frac{\mathrm{Cov}(X,Y)}{\sigma_X \sigma_Y}. \tag{1.5}$$

- $+1$: perfect positive relationship
- 0: no linear relationship
- -1: perfect negative relationship

Machine learning uses correlation to select and weight relevant features.

1.3 Probability Theory Basics

Machine learning models predict probabilities. Even regression models assume probabilistic noise.

1. Joint Probability — "two events happening together"

Joint probability describes the probability of two events occurring at the same time.

For events A and B:

$$P(A,B) = P(A \cap B).$$

Intuition. Joint probability answers the question: *"What is the probability that A happens **and** B happens?"*

In Machine Learning.

- Generative models learn the full joint distribution $P(x,y)$, which allows them to generate new data and compute conditional probabilities.
- Naive Bayes uses the joint distribution in a simplified form, computing $P(x,y)$ under conditional independence assumptions between features.

2. Conditional Probability — "probability of A given that B is true"

Conditional probability formalizes how one event influences another:

$$P(A \mid B) = \frac{P(A,B)}{P(B)}.$$

Intuition. Think of "filtering the world" to only the situations where B is true. Then ask: *"If we restrict attention to situations where B happens, how often do we see A as well?"*

In Machine Learning.

- Classification models learn $P(y \mid x)$.
- Regression assumes a probabilistic noise model for $P(y \mid x)$.
- Bayesian models update beliefs through conditional probabilities.

Conditional probability expresses *relationships, correlations, and dependencies* between variables.

3. Marginal Probability — "probability of an event ignoring everything else"

Marginalization means "summing out" or "integrating out" variables we do not care about.

Discrete case:

$$P(A) = \sum_b P(A, b).$$

Continuous case:

$$P(A) = \int P(A, b)\, db.$$

Intuition. Marginal probability represents the *overall tendency* of an event. It ignores other variables by summing or integrating over them.

In Machine Learning.

- In Bayesian inference, the denominator $P(B)$ in Bayes' theorem is a marginal probability.
- In generative models, marginalization is needed to compute likelihoods.
- In hidden-variable models (e.g., latent factor models, mixture models), marginalization is essential.

Example (discrete).

$$P(\text{rain}) = P(\text{rain}, \text{cold}) + P(\text{rain}, \text{warm}).$$

This sums over all the ways rain can occur, regardless of temperature.

1.4 Bayes' Theorem: Updating Beliefs from Data

1.4.1 Formula

Bayes' theorem describes how we update our belief about an event A after observing some evidence B:

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}. \tag{1.6}$$

What does this mean?

Bayes' theorem tells us:

$$\textbf{Posterior} = \textbf{Likelihood} \times \textbf{Prior} \quad / \quad \textbf{Evidence}$$

More explicitly:

Key Idea

$$P(A \mid B) = \underbrace{P(B \mid A)}_{\text{How well } A \text{ explains the data}} \underbrace{P(A)}_{\text{What we believed before}} \Big/ \underbrace{P(B)}_{\text{How expected the data is overall}}.$$

- $P(A)$ [**prior**]: our initial belief about the hypothesis A before observing any data.
- $P(B \mid A)$ [**likelihood**]: how well hypothesis A explains or predicts the observed data B.
- $P(B)$ [**evidence** or **marginal likelihood**]: the total probability of seeing the data B under *all* possible hypotheses.
- $P(A \mid B)$ [**posterior**]: our updated belief about A after incorporating the information from B.

Intuition

Bayes' theorem answers the question:

"Given what I just observed, how should I update what I believe?"

It combines two ideas:

1. **What we believed before** (the prior)
2. **How surprising the data is under each hypothesis** (the likelihood)

The evidence $P(B)$ ensures everything stays a valid probability distribution (sums to 1).

Why this matters in Machine Learning

Bayes' theorem is the foundation of: Naive Bayes classifier, Bayesian Linear Regression, Bayesian Neural Networks, All probabilistic and generative models.

It formalizes **learning from data**. We start with a belief (prior), then see data, and update our belief (posterior).

1.5 Likelihood

The concept of **likelihood** answers a very important question in statistics and machine learning:

Key Idea

"If this parameter θ were true, how well would it explain the data I observed?"

This is different from asking about the probability of future events. Likelihood is specifically about evaluating how plausible a parameter is, given the data we already have.

Definition

For a dataset $x = (x_1, x_2, \ldots, x_n)$ and a model with parameter θ, the likelihood is defined as:

$$L(\theta \mid x) = P(x \mid \theta)$$

This expression should be read as:

> **"Given θ, what is the probability of observing the dataset x?"**

Interpretation

- $x = (x_1, x_2, \ldots, x_n)$: the observed data. These values are **fixed**. We already saw them.
- θ: the parameter of the model. This is **unknown** and is what we want to estimate.
- $P(x \mid \theta)$: the model's prediction about how likely the data is, if θ were the true parameter.

Important Distinction: Probability vs. Likelihood

- **Probability:** θ is fixed, data is random. (Used to predict future observations.)
- **Likelihood:** data is fixed, θ is variable. (Used to evaluate or estimate parameters.)

This is a subtle but essential idea in statistical learning.

The Likelihood Function

Note

The likelihood function is simply the probability of the data, viewed as a function of the parameter:

$$L(x; \theta) = P(x \mid \theta)$$

It tells us how much support the data gives to each possible value of θ.

"Better-fitting parameters give higher likelihood."

Crucially: likelihood is *not* a probability distribution over θ. It does **not** integrate to 1. It is just a scoring function that says which values of θ explain the data best.

The i.i.d. Assumption

In most machine learning models, the data points are assumed to be:

- **independent:** knowing one datapoint tells nothing about another,
- **identically distributed:** all datapoints come from the same distribution.

Under this assumption:

$$L(x \mid \theta) = \prod_{i=1}^{n} P(x_i \mid \theta)$$

Why a product?

Because the joint probability of independent events equals the product of their individual probabilities.

This is a key simplification that makes likelihood computation possible for large datasets.

Summary

- Likelihood measures how well a model with parameter θ explains the observed data.
- It is the central tool for parameter estimation.
- Almost all ML algorithms rely on likelihood:
  - Linear Regression
  - Logistic Regression
  - Neural Networks
  - Decision Trees and Random Forests (implicitly)
  - XGBoost
- Training a model often means:

$$\text{maximize likelihood} \quad \text{or} \quad \text{minimize negative log-likelihood.}$$

In short, likelihood connects your model to your data. It is the mathematical backbone of statistical learning.

1.6 Probability Distributions: Shape of Data

A probability distribution describes **how data behaves**. It tells us which values are likely, which are rare, and what kind of uncertainty or randomness we should expect in the data.

Machine learning models often assume certain distributions because they determine the model's behavior and how it handles noise.

1.6.1 Discrete Distributions

Discrete distributions describe random variables that take values from a finite or countable set (e.g., 0, 1, 2, 3, ...).

Bernoulli Distribution A Bernoulli variable represents a single binary outcome:

$$P(X = x) = p^x(1-p)^{1-x}, \qquad x \in \{0,1\}. \tag{1.7}$$

Interpretation:

- Models "success or failure" events.
- Examples: coin flip, email is spam/not spam, churning/not churning customer.
- Parameter p is the probability of success.

Binomial Distribution The binomial distribution models the number of successes in n independent Bernoulli trials:

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}. \tag{1.8}$$

Interpretation:

- Repeating the same experiment n times.
- Examples: number of heads in n coin flips, number of defective items in a batch.
- Gives the distribution of the *count of successes*.

Poisson Distribution The Poisson distribution models the number of events occurring in a fixed time or space interval:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}. \tag{1.9}$$

Interpretation:

- Used for rare events happening unpredictably.
- Examples: number of incoming calls per minute, number of website hits per second, number of accidents per day.
- Parameter λ is both the mean and the variance.

1.6.2 Continuous Distributions

Continuous distributions describe variables that take real-number values (e.g., height, temperature, time).

Gaussian (Normal Distribution)

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{1.10}$$

Interpretation:

- The most important distribution in statistics.
- Data clusters around a mean μ with spread σ.
- Many ML models assume Gaussian noise.
- Examples: measurement error, natural variations, height.

Exponential Distribution

$$f(x) = \lambda e^{-\lambda x} \tag{1.11}$$

Interpretation:

- Models the time between independent random events.
- Examples: time until next earthquake, time until customer arrival.
- Memoryless property: past values do not change future probability.

Multivariate Gaussian

$$f(x) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right) \tag{1.12}$$

Interpretation:

- A Gaussian distribution in multiple dimensions.
- μ is a mean vector, Σ is a covariance matrix.
- Models correlated variables.
- Used heavily in:
  - PCA (Principal Component Analysis)
  - Gaussian Mixture Models (GMMs)
  - Bayesian inference

Why Distributions Matter in Machine Learning

Different distributions encode different assumptions about uncertainty:

- Bernoulli / Binomial → binary or count data
- Gaussian → continuous data with natural noise
- Poisson → rare event counts
- Exponential → waiting times
- Multivariate Gaussian → correlated continuous data

Choosing an appropriate distribution helps us build models that accurately reflect the data-generating process.

1.7 Statistical Inference: Learning Parameters From Data

Statistical inference is about using data to learn the best values for the parameters of a model. In machine learning, almost every algorithm can be understood as an inference method.

We focus on two fundamental ideas:

- **Maximum Likelihood Estimation (MLE)**
- **Maximum A Posteriori Estimation (MAP)**

These appear throughout regression, classification, probabilistic models, and even deep learning.

1.7.1 Maximum Likelihood Estimation (MLE)

MLE answers a simple question:

Key Idea

"Which parameter value makes the observed data the most likely?"

If $x = (x_1, \ldots, x_n)$ is the observed data and θ is an unknown parameter, the MLE estimate is [1]:

$$\hat{\theta}_{\text{MLE}} = \arg\max_{\theta} L(\theta \mid x) \tag{1.13}$$

where

$$L(\theta \mid x) = P(x \mid \theta)$$

is the likelihood of the data under the parameter θ.

[1] max → what is the highest value. argmax → where is the highest value

Why do we take the log?

Because products of probabilities become sums:

$$\hat{\theta}_{\text{MLE}} = \arg\max_{\theta} \log L(\theta \mid x)$$

This is numerically stable and makes optimization easier.

Intuition

- The data x is considered fixed.
- The parameter θ varies.
- We choose the θ that best *explains* the data.

This idea lies behind most ML algorithms: Linear regression (least squares), Logistic regression, Naive Bayes, Neural networks (via cross-entropy).

All of them optimize a likelihood or its negative log.

1.7.2 Maximum A Posteriori Estimation (MAP)

MAP estimation is the Bayesian version of learning parameters.

Instead of asking:

"Which parameter makes the data most likely?"

we ask:

Key Idea

"Which parameter is most plausible, given the data and my prior beliefs?"

The MAP estimate is:

$$\hat{\theta}_{\text{MAP}} = \arg\max_{\theta} P(\theta \mid x) \tag{1.14}$$

Using Bayes' theorem:

$$P(\theta \mid x) = \frac{P(x \mid \theta)\, P(\theta)}{P(x)}$$

Since $P(x)$ does not depend on θ, we ignore it in maximization:

$$\hat{\theta}_{\text{MAP}} = \arg\max_{\theta} \left[\log P(x \mid \theta) + \log P(\theta)\right]$$

Interpretation

- **MLE**: only cares about the data.
- **MAP**: cares about data *and* prior knowledge.

Example

Examples of priors:
Priors express what we believe about the parameter θ before seeing data. In machine learning, they often appear naturally as regularization.
1. Prior: "Parameters should be small"
This corresponds to a Gaussian prior:

$$P(\theta) \propto \exp\left(-\theta^2\right),$$

which leads to the penalty θ^2 after taking $-\log$. This is exactly **L2 regularization**.
2. Prior: "Many parameters should be zero"
This corresponds to a Laplace prior:

$$P(\theta) \propto \exp(-|\theta|),$$

which leads to the penalty $|\theta|$. This is **L1 regularization**, which encourages sparsity.

Connection between MLE and MAP

$$\text{Posterior} \propto \text{Likelihood} \times \text{Prior}$$

- If the prior is flat (uninformative), MAP = MLE.
- If the prior is strong, MAP pulls the estimate toward the prior belief.

MAP can be seen as:

$$\text{MAP} = \text{MLE} + \text{regularization}.$$

This is why many ML algorithms with regularization (ridge, lasso, weight decay) are actually MAP estimators in disguise.

1.8 Linear Regression: Statistical Interpretation

Linear regression is one of the simplest and most fundamental models in machine learning. It describes a relationship between input features and a continuous output.

Model

We assume the data is generated according to:

$$y = X\beta + \varepsilon \tag{1.15}$$

- X is the matrix of input features.
- β is a vector of unknown parameters.
- ε is random noise (unpredictable variation).

The key probabilistic assumption:

$$\varepsilon \sim \mathcal{N}(0, \sigma^2 I)$$

This means:

- noise is Gaussian,
- noise has zero mean,
- noise is independent with constant variance.

With this assumption, each y_i is normally distributed around $X_i\beta$.

Least Squares Estimation

The most common way to estimate β is to minimize the sum of squared errors:

$$\hat{\beta} = \arg\min_{\beta} \|y - X\beta\|^2$$

This objective measures how well the model's predictions match the data.

The minimizer has a closed-form solution:

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

This solution exists when $X^T X$ is invertible.

Probabilistic Interpretation

Ordinary Least Squares (OLS) is not only a geometric method—it is also a probabilistic one.

If the noise is Gaussian, then the likelihood of observing y given β is:

$$P(y \mid X, \beta) = \frac{1}{(2\pi\sigma^2)^{n/2}} \exp\left(-\frac{1}{2\sigma^2}\|y - X\beta\|^2\right)$$

Maximizing this likelihood is equivalent to minimizing the squared error.

Thus:

$$\text{OLS} = \text{MLE under Gaussian noise.}$$

Interpretation:

- Linear regression assumes data is generated as a linear trend + Gaussian noise.
- Fitting the model means choosing β that makes the data most likely.
- This gives a deep statistical foundation to the method.

Linear regression is therefore both:

$$\text{a geometric projection problem} \quad \text{and} \quad \text{a probabilistic parameter estimation problem..}$$

Key Idea

What is linear regression?
Linear regression is a simple model that tries to describe a relationship between input variables (features) and an output variable by fitting a straight line (or a hyperplane in higher dimensions).

- You give the model input data X and outputs y.
- The model tries to find parameters β so that the prediction $X\beta$ is close to y.
- The assumption is that the underlying true relationship is approximately *linear*.

Geometric view:

- The model projects y onto the space spanned by the columns of X.
- The result is the "best-fitting line" in the least-squares sense.

Statistical view:

- The data is assumed to follow
$$y = X\beta + \varepsilon,$$
where ε is random noise with Gaussian distribution.

- Fitting the model means choosing β that makes the observed data most probable.

Linear regression is a fundamental tool because it is easy to interpret, computationally efficient, and the foundation of more advanced models.

1.9 Logistic Regression: Probabilistic Classification

Linear regression predicts a continuous outcome. Logistic regression adapts the idea to predict *probabilities* for binary outcomes (0 or 1).

Model

We model the probability of the outcome being 1 as:

$$P(y = 1 \mid x) = \sigma(w^T x) \tag{1.16}$$

using the sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

Properties of the sigmoid:

- outputs values between 0 and 1,
- smoothly increases, S-shaped,
- interpretable as a probability.

This means logistic regression predicts:

"How likely is it that $y = 1$ given the input x?"

Log-Likelihood

Assume each data point (x_i, y_i) is drawn independently. The likelihood of the entire dataset under parameters w is:

$$\ell(w) = \sum_{i=1}^{n} \left[y_i \log \sigma(w^T x_i) + (1 - y_i) \log\left(1 - \sigma(w^T x_i)\right) \right]$$

This is the log-likelihood of the Bernoulli model.

Training = maximize the log-likelihood.

This chooses the w that makes the observed labels most probable.

Connection to Cross-Entropy Loss

Instead of maximizing $\ell(w)$, we minimize its negative:

$$\mathscr{L}(w) = -\ell(w)$$

This is the **cross-entropy loss**, the most widely used loss in classification and neural networks.

Interpretation:

- If the model assigns a high probability to the true label, the loss is small.
- If the model assigns a low probability to the true label, the loss is large.

Thus training logistic regression is about making correct labels more probable.

Summary

Logistic regression is:

- a probabilistic model for binary classification,
- trained via maximum likelihood,
- equivalent to minimizing cross-entropy,
- foundational for neural networks, softmax classifiers, and deep learning.

It takes the linear model $w^T x$ and turns it into a probability through the sigmoid function.

1.10 Bias–Variance Tradeoff: Why Models Fail

When we train a model, we want its predictions $\hat{f}(x)$ to be close to the true output y. A key question in machine learning is:

Why do models make errors, even after training?

The answer is given by the bias–variance decomposition:

$$\mathbb{E}[(y - \hat{f}(x))^2] = \underbrace{\text{Bias}[\hat{f}(x)]^2}_{\text{model too simple}} + \underbrace{\text{Var}[\hat{f}(x)]}_{\text{model too complex}} + \sigma^2_{\text{noise}}. \tag{1.17}$$

This formula says that prediction error comes from *three* different sources.

1. Bias: Error from Wrong Assumptions

Bias measures how far the model's average prediction is from the true relationship.

- High bias means the model is too simple.
- It cannot capture the true pattern.
- It makes the same mistakes no matter how much data we have.

Examples:

- Using a straight line to fit a quadratic curve.
- A linear model for data with strong nonlinear structure.

This is called **underfitting**.

2. Variance: Error from Sensitivity to Data

Variance measures how much predictions change if we train on a different dataset.

- High variance means the model is too flexible.
- It memorizes random noise in the training data.
- Small changes in the data lead to large changes in the model.

Examples:

- A decision tree that grows too deep.
- A neural network trained with too few data points.

This is called **overfitting**.

3. Irreducible Noise

$$\sigma^2_{\text{noise}}$$

This is randomness in the data that no model can ever explain.

Examples:

- Measurement error.
- Natural variability in human behavior.
- Random fluctuations in physical or economic systems.

Even a perfect model cannot reduce this part of the error.

Why This Matters

The bias–variance tradeoff explains:

- **Underfitting**: high bias, low variance.
- **Overfitting**: low bias, high variance.
- **Regularization**: reduces variance by simplifying the model.
- **Cross-validation**: detects overfitting by testing on unseen data.
- **Ensembles**: average predictions to reduce variance.

> **The goal of learning is to find a model with a good balance:** low bias *and* low variance.

Example-Recap

Understanding Probability, Likelihood, MLE, MAP

We have a coin, and we toss it $n = 10$ times. We observe:

$$x = \text{H H T H H T H H H T} \qquad \Rightarrow \qquad k = 7 \text{ heads, } 3 \text{ tails.}$$

Let $\theta = P(\text{Heads})$ be the (unknown) probability that the coin lands Heads.

1. Probability: parameter fixed, data random

Here we assume that the coin bias *is already known*. Suppose:

$$\theta = 0.6.$$

Then we ask:

"If the coin has $\theta = 0.6$, what is the probability of getting exactly 7 heads in 10 tosses?"

This is a binomial probability:

$$P(X = 7 \mid \theta = 0.6) = \binom{10}{7} (0.6)^7 (0.4)^3.$$

Let us break down each part:

- $\binom{10}{7}$ counts how many sequences of 7 heads and 3 tails exist.
- $(0.6)^7$ is the probability of getting Heads 7 times.
- $(0.4)^3$ is the probability of getting Tails 3 times.

A numerical evaluation:

$$\binom{10}{7} = 120, \quad (0.6)^7 \approx 0.02799, \quad (0.4)^3 = 0.064.$$

Thus:

$$P(X = 7 \mid \theta = 0.6) = 120 \times 0.02799 \times 0.064 \approx 0.215.$$

So if $\theta = 0.6$, this data occurs with probability about 21.5%.

Here:

$$\theta \text{ fixed}, \quad X \text{ random}.$$

This is the usual "forward" direction in probability.

2. Likelihood: data fixed, parameter varies

Now we switch perspectives completely. We already saw the data:

$$k = 7.$$

We treat the data as fixed. Now we ask:

"For this fixed data (7 Heads), how plausible is each possible value of θ?"

This is the likelihood:

$$L(\theta \mid x) = P(x \mid \theta) = \binom{10}{7} \theta^7 (1-\theta)^3.$$

Key idea:

- The combinatorial factor $\binom{10}{7}$ does **not** depend on θ.
- So the likelihood is shaped by:
$$\theta^7 (1-\theta)^3.$$

Interpretation:

- Data is fixed ($k = 7$).
- θ varies between 0 and 1.
- The likelihood is a *score function*, not a probability distribution over θ.

Probability vs Likelihood:

$$\text{Probability: fix } \theta, \text{ vary data.}$$

$$\text{Likelihood: fix data, vary } \theta.$$

3. Bayesian View: Prior, Likelihood, Posterior

The Bayesian view adds one more ingredient: a **prior** about θ.
Instead of treating θ as an unknown but fixed number, we treat it as a random variable with its own distribution $P(\theta)$ (our belief *before* seeing the data).
Bayes' theorem connects:

$$P(\theta \mid x) = \frac{P(x \mid \theta) P(\theta)}{P(x)}.$$

In this formula:

- $P(\theta)$ is the **prior**: what we believe about θ before seeing data.
- $P(x \mid \theta)$ is the **likelihood**: how well each θ explains the data.
- $P(\theta \mid x)$ is the **posterior**: updated belief about θ after seeing data.
- $P(x)$ is the **evidence**: a normalizing constant to make probabilities sum to 1.

We often write this in proportional form:

$$P(\theta \mid x) \propto P(x \mid \theta) P(\theta).$$

So in words:

Posterior = Likelihood $\times$ Prior (up to a constant).

The likelihood from step 2 is what *transforms* the prior into the posterior.

4. Maximum Likelihood Estimation (MLE)

MLE asks a simple question:

"Which value of θ makes the observed data (7 heads) most likely?"

We take the likelihood:

$$L(\theta \mid x) = \theta^7(1-\theta)^3,$$

and choose the θ that makes this expression as large as possible:

$$\hat{\theta}_{\text{MLE}} = \arg\max_{\theta} \theta^7(1-\theta)^3.$$

For the binomial model, the answer is always:

$$\hat{\theta}_{\text{MLE}} = \frac{k}{n}.$$

Here:

$$\hat{\theta}_{\text{MLE}} = \frac{7}{10} = 0.7.$$

Meaning: MLE simply uses the data. It says:

"You saw heads 70% of the time. So my best estimate is $\theta = 0.7$."

5. Bayesian Inference and MAP

The Bayesian approach adds one more ingredient: a **prior belief** about θ.
Suppose our prior is:

$$\theta \sim \text{Beta}(5,5),$$

which expresses the idea that the coin is probably close to fair ($\theta \approx 0.5$).[a]
After seeing the data (7 heads out of 10), the posterior becomes:

$$\theta \mid x \sim \text{Beta}(12,8).$$

MAP estimate

MAP chooses the value of θ that is most probable under the posterior:

$$\hat{\theta}_{\text{MAP}} = \frac{\alpha' - 1}{\alpha' + \beta' - 2} = \frac{12-1}{12+8-2} = \frac{11}{18} \approx 0.611.$$

Meaning:

- The data pushes the estimate toward 0.7.
- The prior pulls the estimate toward 0.5.

So MAP becomes:

$$\text{something between } 0.5 \text{ and } 0.7,$$

which here is ≈ 0.611.

MAP = "**update your belief by mixing prior + data**".

If the prior were weak (e.g. Beta(1,1)), MAP would be almost the same as MLE.

6. Summary

- **Probability** $P(x \mid \theta)$: given a model, what data is expected?
- **Likelihood** $L(\theta \mid x)$: given data, how plausible is each parameter?
- **MLE**: maximize likelihood (data only).
- **Bayesian Update**: Posterior $\propto$ Likelihood $\times$ Prior.
- **MAP**: maximize posterior = MLE + prior information.

---

[a] **The Beta distribution and the meaning of α and β.**

The *Beta distribution* is a probability distribution on the interval $[0,1]$, commonly used as a prior for unknown probabilities such as the bias θ of a coin. Its density is

$$f(\theta \mid \alpha,\beta) = \frac{1}{B(\alpha,\beta)}\,\theta^{\alpha-1}(1-\theta)^{\beta-1}, \qquad 0 \le \theta \le 1,$$

where the normalising constant

$$B(\alpha,\beta) = \int_0^1 t^{\alpha-1}(1-t)^{\beta-1}\,dt = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}$$

is the *Beta function*.

A useful interpretation is:

- $\alpha - 1$ acts like a prior count of heads,
- $\beta - 1$ acts like a prior count of tails.

Thus a Beta(α,β) prior encodes the belief that, before seeing any real data, we have already observed $(\alpha-1)$ heads and $(\beta-1)$ tails.

The Beta distribution is a conjugate prior for the binomial likelihood. If $\theta \sim \text{Beta}(\alpha,\beta)$ and we observe k heads in n flips, then the posterior is

$$\theta \mid x \sim \text{Beta}(\alpha+k,\ \beta+n-k),$$

so Bayesian updating simply adds the observed counts to the prior counts.

2

Python for Machine Learning

In this chapter we will discuss the basic conceptions for the core of Machine Learning of supervised data, loss function and risk minimazation. Also we will present the Python workflow we need.

2.1 Supervised Data

Supervised learning refers to a setting where we observe a collection of examples, each consisting of:

- an **input** (also called *features*), and
- an **output** (also called *label* or *target*).

The goal is to learn a rule that maps inputs to outputs so that we can make accurate predictions for new, unseen data.

Key Idea

"We show the model many examples of input → correct output,
and we ask it to learn the pattern so it can predict the output for new inputs."

2.1.1 The Form of Supervised Data

We are given a dataset consisting of n labeled examples:

$$D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}.$$

Each pair (x_i, y_i) contains:

$$x_i \in \mathbb{R}^d \quad \text{(a feature vector with } d \text{ components)},$$

$$y_i \quad \text{(a target value we want to predict)}.$$

The target variable may be:

- **Real-valued** ($y_i \in \mathbb{R}$) → **regression**

- **Categorical** ($y_i \in \{0,1\}$ or $y_i \in \{1,\dots,K\}$) $\rightarrow$ **classification**

In supervised learning, the dataset includes both inputs and correct outputs. This is what distinguishes it from **unsupervised** learning, where we only have the inputs x_i and no labels.

Example

Example: House Price Dataset

- $x_i = (\texttt{size}, \texttt{number\_of\_rooms}, \texttt{age\_of\_house}) \in \mathbb{R}^3$
- $y_i = \texttt{price}$ of the house

Here:

- Each (x_i, y_i) is one house in the dataset.
- Supervised learning asks: *"Given the features of a new house, can we predict its price?"*
- Unsupervised learning would only have x_i and would ask: *"Do houses naturally form groups or patterns?"*

2.1.2 The Objective of Supervised Learning

We assume there exists an unknown function f^* that relates inputs to outputs:

$$y = f^*(x) + \varepsilon, \tag{2.1}$$

where:

- f^* is the **true** (but unknown) relationship,
- ε is random **noise** or unpredictable variation.

A machine learning model builds an approximation $f_\theta(x)$, parameterized by θ:

$$\hat{y} = f_\theta(x). \tag{2.2}$$

The goal of supervised learning is to find parameters θ such that $f_\theta(x)$ approximates $f^*(x)$ as closely as possible.

Key Idea

Think of f^* vs. f_θ:

- f^*: the ideal rule that nature is using (we never see it exactly).
- f_θ: our model's best attempt to imitate this rule using data.

Training = adjusting θ so that $f_\theta(x)$ behaves like $f^*(x)$ on the data we have.

2.1.3 Regression vs. Classification

Supervised learning problems fall into two main categories.

Regression The target is continuous:

$$y \in \mathbb{R}.$$

Examples: predicting house prices, estimating temperature, forecasting demand or sales.

Classification The target is discrete:

$$y \in \{0,1\} \quad \text{(binary)}, \qquad y \in \{1,\dots,K\} \quad \text{(multi-class)}.$$

Examples: spam vs. not spam, image classification (cat / dog / car / ...), medical diagnosis (disease present / not present).

Note

Same input, different task:
If we use patient data (age, blood pressure, etc.) to predict *blood sugar level* $\Rightarrow$ regression.
If we use the same data to predict *diabetic vs. not diabetic* $\Rightarrow$ classification.
The data can be the same, but the **type of target** changes the learning problem.

2.1.4 i.i.d. Assumption

Supervised learning typically assumes that the samples

$$(x_1, y_1), \dots, (x_n, y_n)$$

are **i.i.d.** (independent and identically distributed):

- **Independent:** Each example is collected independently of the others. One data point does not influence another (e.g. one customer's purchase does not change another customer's features).
- **Identically distributed:** All samples come from the same underlying distribution (same population, same data-generating process).

Machine learning works only if the model is trained and tested on data that follow the *same distribution*. If this holds, then:

$$\text{training data} \approx \text{future data}$$

so good performance on training/validation data tells us something meaningful about performance on unseen data.

Key Idea

Why i.i.d. is important:
i.i.d. is important because it guarantees that the data used for training, validation, and testing all come from the same underlying distribution. If this holds, then good performance on training/validation data is informative about performance on future data. If this does not hold (distribution shift), a model may perform badly even if it seemed to work well during training.

The i.d.d, assumption underlies key ML methods such as: empirical risk minimization, maximum likelihood estimation, and cross-validation.

2.1.5 Features and Target Variables

A feature vector can be written as:

$$x_i = (x_{i1}, x_{i2}, \ldots, x_{id})^\top.$$

Types of features include:

- **Numerical:** real numbers (e.g. age, income, temperature)
- **Categorical:** categories (e.g. country, color) encoded using one-hot or label encoding
- **Ordinal:** ordered categories (e.g. low / medium / high)
- **Boolean:** true/false (0 or 1)
- **Derived:** new features created from raw data (e.g. BMI from height and weight, room_per_person, etc.)

Example

Example: Feature Vector for a House

$$x_i = (\text{size}, \text{rooms}, \text{age}, \text{distance\_to\_center})^\top = (85, 3, 20, 4.5)^\top$$

If we also know the house price y_i, the pair (x_i, y_i) becomes one supervised example in our dataset.

A machine learning model uses these features to learn patterns that generalize well to new, unseen data.

2.1.6 The Learning Goal

Given supervised data, we seek a function $f_\theta(x)$ that:

- fits the training data well,
- is not overly complex (to avoid overfitting),
- performs well on new, unseen data.

Key Idea

Core tension in supervised learning:

- If the model is too simple → it cannot capture the pattern (underfitting).
- If the model is too complex → it memorizes noise (overfitting).

The art of machine learning is to find a model and training procedure that *captures the signal* in the data while ignoring as much noise as possible.

2.2 Loss Function and Risk Minimization

In supervised learning, a model produces predictions $\hat{y} = f_\theta(x)$. To measure how good or bad these predictions are, we use a **loss function**.

A **loss function** is a mathematical rule that measures the error between the predicted value $\hat{y}$ and the true target y[1]:

$$L(y, \hat{y}) \in \mathbb{R}_{\geq 0}. \tag{2.3}$$

A small loss means the prediction is good; a large loss means it is bad.

Note

Meaning of the symbols:

- y: the **true target value** from the dataset.
- $\hat{y} = f_\theta(x)$: the **model's prediction**.
- $L(y, \hat{y})$ or $\ell(y, \hat{y})$: the **loss for one example**, measuring how wrong the prediction is.

[1] Important notation distinction:

- $L(y, \hat{y})$ (capital L): the **loss function** for a single example. It outputs a non-negative number that measures how wrong a prediction is.
- $\ell(y, \hat{y})$ (lowercase ℓ): simply an *alternative notation* for the same per-example loss used by many textbooks. In this chapter, L and ℓ both refer to *single-example loss*.
- $L(\theta)$ (capital L applied to parameters): the **total loss** or **empirical risk**, i.e., the average loss over the entire dataset.

So:

$$\ell(y_i, \hat{y}_i) = L(y_i, \hat{y}_i) \quad \text{(single-example loss)},$$

while

$$L(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, \hat{y}_i) \quad \text{(aggregate loss)}.$$

In the previous chapter, the symbol $L(\theta \mid x)$ denoted the *likelihood function*. Here, $L(y, \hat{y})$ denotes the *loss function*.

Training a model means: *make losses as small as possible*.

2.2.1 Loss for a Single Example

For one training example (x_i, y_i), the loss is:

$$L(y_i, f_\theta(x_i)).$$

This quantifies the prediction error for that single datapoint.

Common choices:

- **Squared loss (regression)**
$$L(y, \hat{y}) = (y - \hat{y})^2.$$
- **Absolute loss**
$$L(y, \hat{y}) = |y - \hat{y}|.$$
- **Cross-entropy loss (classification)**
$$L(y, \hat{p}) = -\left[y \log(\hat{p}) + (1-y)\log(1-\hat{p})\right],$$
where $\hat{p}$ is the predicted probability of the positive class.

Different loss functions produce different learning behaviours.

2.2.2 Empirical Risk: Total Loss on the Dataset

Because we cannot compute the true risk, we estimate it by averaging the loss over the training data. This gives the **empirical risk**:

$$R_{\text{emp}}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell\big(y_i, f_\theta(x_i)\big). \tag{2.4}$$

Here, $\ell(y_i, f_\theta(x_i))$ is the loss for a *single* example, and $R_{\text{emp}}(\theta)$ is the average loss over all n examples.

Different tasks use different losses:

- regression:
$$\ell(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2,$$
- classification:
$$\ell(y_i, \hat{y}_i) = -[y_i \log \hat{p}_i + (1-y_i)\log(1-\hat{p}_i)],$$
- robust regression:
$$\ell(y_i, \hat{y}_i) = |y_i - \hat{y}_i|.$$

Note

The empirical risk is the **average error over all training samples**. It is the quantity we can compute in practice.

2.2.3 Risk Minimization

Training a model means choosing parameters θ that minimize empirical risk:

$$\hat{\theta} = \arg\min_{\theta} R_{\text{emp}}(\theta). \tag{2.5}$$

This is called **Empirical Risk Minimization (ERM)**.

Key Idea

ERM = "Choose the parameters that give the lowest average loss on the training data."

Every major supervised learning algorithm (linear regression, logistic regression, trees, SVMs, neural networks) follows this principle.

Note

Ideally, we want to minimize the *true* risk:

$$R(\theta) = \mathbb{E}_{(x,y)\sim P_{\text{data}}}[\ell(y, f_\theta(x))].$$

But we do not know the true distribution P_{data}. So we minimize R_{emp} instead.[a]
Generalization methods (regularization, cross-validation) help us ensure that minimizing empirical risk also reduces true risk.

[a] The symbol $\mathbb{E}$ denotes the **expectation** (average value) taken over the true but unknown data distribution P_{data}. In words:

$$\mathbb{E}_{(x,y)\sim P_{\text{data}}}[\ell(y, f_\theta(x))] = \text{the average loss we would obtain on all possible data points.}$$

Since we do not know P_{data}, we approximate this expectation using the empirical average over the training dataset.

2.2.4 Example: Linear Regression Loss

Example

Goal: Predict a real number (e.g., house price).
Linear regression assumes the model predicts using a linear function:

$$\hat{y}_i = w^\top x_i.$$

Here:

- x_i is the feature vector of example i,

- w is the vector of parameters (weights),
- $\hat{y}_i$ is the model's predicted value.

Loss for one example.
We measure how wrong the prediction is using the squared error:

$$\ell_i = (y_i - w^\top x_i)^2.$$

Interpretation:

- If prediction $\hat{y}_i$ is close to the true value y_i, the loss is small.
- If the prediction is far from y_i, the loss becomes large (because of the square).

Empirical risk (total loss) is the average squared error over all examples:

$$R_{\text{emp}}(w) = \frac{1}{n}\sum_{i=1}^{n}(y_i - w^\top x_i)^2.$$

Training linear regression means:

Find the weights w that make the average squared error as small as possible.

This is classical *least squares*.

2.2.5 Example: Logistic Regression Loss

Example

Goal: Predict a binary outcome (0 or 1). Example: tumour is malignant (1) or benign (0).
Logistic regression predicts a *probability*:

$$\hat{p}_i = \sigma(w^\top x_i),$$

where $\sigma(z)$ is the sigmoid function, which always returns a value in $(0,1)$.
Interpretation:

- $w^\top x_i$ is a linear score.
- $\sigma(w^\top x_i)$ converts that score into a probability.
- $\hat{p}_i$ is the model's belief that the correct label is 1.

Loss for one example.
We use the *cross-entropy* loss:

$$\ell_i = -\left[y_i \log(\hat{p}_i) + (1-y_i)\log(1-\hat{p}_i)\right].$$

Pedagogical meaning:

- If $y_i = 1$ and the model predicts $\hat{p}_i$ close to 1, then $\log(\hat{p}_i)$ is close to 0 (good), so loss is small.
- If $y_i = 1$ but the model predicts $\hat{p}_i$ close to 0, then $\log(\hat{p}_i)$ becomes very negative, so loss becomes large.
- If $y_i = 0$, the loss behaves symmetrically using $\log(1 - \hat{p}_i)$.

Thus:

Cross-entropy loss penalizes confident wrong predictions very strongly.

This loss has a deep connection to statistics:

$$\text{Minimizing cross-entropy} \quad \Longleftrightarrow \quad \text{Maximizing the log-likelihood of a Bernoulli model.}$$

In other words, logistic regression is a probabilistic model trained with MLE.

2.2.6 Regularization: Controlling Model Complexity

Minimizing empirical risk alone often leads to **overfitting**: the model fits the training data too closely and performs poorly on new data.

To prevent this, we add a **regularization term** that penalizes overly complex models:

$$\hat{\theta} = \arg\min_{\theta} \Big[R_{\text{emp}}(\theta) + \lambda\, \Omega(\theta) \Big]. \tag{2.6}$$

- $\lambda > 0$ controls the strength of regularization (large λ = stronger penalty).
- $\Omega(\theta)$ is a measure of model complexity.

Common choices:

- **L2 regularization (Ridge):**
$$\Omega(\theta) = \|\theta\|_2^2$$
  - discourages large parameter values,
  - smooths the model,
  - corresponds to a Gaussian prior in MAP.
- **L1 regularization (Lasso):**
$$\Omega(\theta) = \|\theta\|_1$$
  - encourages sparsity (many parameters become exactly zero),
  - performs implicit feature selection,
  - corresponds to a Laplace prior in MAP.

Regularization balances two goals:

- **fit the data** (low empirical risk),
- **remain simple enough** to generalize (low complexity).

2.2.7 Summarize

To sum up, until now:

- A **loss function** measures prediction error for a single example.
- **Risk** extends this idea over the whole data distribution.
- Because the true distribution is unknown, we minimize the **empirical risk** on the training set.
- To avoid overfitting, we add a **regularization term** that penalizes overly complex models.

Altogether, most machine learning algorithms solve an optimization problem of the form:

$$\theta^* = \arg\min_{\theta} \left[\frac{1}{n} \sum_{i=1}^{n} L(y_i, \hat{y}_i) + \lambda \, \Omega(\theta) \right].$$

This framework is the foundation of nearly all modern machine learning methods, from linear and logistic regression to support vector machines and deep learning.

2.3 Train / Validation / Test

When we train a model, we minimize the empirical risk on the training set:

$$R_{\text{emp}}(\theta) = \frac{1}{n} \sum_{i=1}^{n} L\big(y_i, f_\theta(x_i)\big).$$

However, a model can achieve very low loss on the training data and still perform poorly on new data. This is **overfitting**: the model memorizes the training data instead of learning general patterns.

Key Idea

Why split the data?
A model may perform very well on the data it was trained on simply because it has *memorized* it. To detect and prevent this overfitting, we must always evaluate the model on data that it has *never seen before*. Only then can we estimate how well it will perform on truly new, future data.

2.3.1 The Three Splits

1. Training Set

- Used to learn the model parameters θ.

- Examples:
  - Linear regression learns weights β.
  - Random Forest learns tree splits.
  - Neural networks learn millions of parameters via gradient descent.
- The model updates its parameters **only** using this data.

Note

Training = fitting the parameters. The model tries to minimize the loss on this set.

2. Validation Set The validation set is **not** used to update the model's parameters. Instead, it evaluates how design choices affect performance.

Used for:

- hyperparameter tuning (learning rate, regularization λ, tree depth, ...)
- selecting the best model architecture
- early stopping in neural networks

Parameters vs. Hyperparameters

- **Parameters** learned directly from training data (weights, coefficients, thresholds)
- **Hyperparameters** chosen using the validation set (learning rate, regularization strength, number of layers, max depth)

Example

Example: Try several models with different λ values. Choose the one with the lowest *validation loss*.

3. Test Set The test set provides the **final, unbiased** evaluation.

- Used only once, after all training and hyperparameter tuning is complete.
- Simulates real-world, future data.
- Must never be used to make design decisions.

Note

Never leak information from the test set. If the test set influences training decisions, the performance estimate becomes invalid.

Summarise

A typical split:

70% train | 15% validation | 15% test

Workflow:

1. **Train set:** Used to fit the model's parameters. The model learns patterns directly from this data.
2. **Validation set:** Used to tune hyperparameters and compare different model choices. It helps select the version of the model that generalizes best.
3. **Test set:** Used only at the very end, after all decisions are made. It provides an unbiased estimate of the model's final performance on new, unseen data.

2.3.2 Data Leakage

A crucial rule:

The test set must remain completely unseen.

Examples of leakage:

- normalizing using the full dataset instead of the training set,
- choosing hyperparameters using the test set,
- extracting features using all data (e.g., PCA on the whole dataset).

These mistakes artificially inflate performance and break generalization.

k-Fold Cross-Validation

When the dataset is small, we cannot afford to hold out a large validation set. Instead, we use **k-fold cross-validation**.

Procedure:

1. Split the dataset into k equal folds.
2. For each fold:
   - train on $k-1$ folds,
   - validate on the remaining fold.

3. Average the validation results.

Typical choices: $k = 5$ or $k = 10$.

Cross-validation is essential for:

- robust hyperparameter tuning,
- model selection,
- reducing variance in evaluation,
- avoiding overfitting on a single validation set.

Summary

- **Training set:** learn parameters.
- **Validation set:** tune hyperparameters and choose the model.
- **Test set:** final, unbiased evaluation of generalization.
- **Cross-validation:** a stable validation method when data is limited.

Splitting the data correctly is essential to ensure that a model truly generalizes and is not simply memorizing the training examples.

2.4 Python Ecosystem

Modern machine learning relies on a small collection of powerful Python libraries that provide efficient numerical computation, data handling, visualization, and ready-to-use learning algorithms. In this section, we introduce the essential tools that will be used throughout the rest of this textbook.

2.4.1 Core Scientific Libraries

NumPy

NumPy provides support for:

- multidimensional arrays,
- vectorized operations,
- random number generation,
- linear algebra routines.

Most machine learning algorithms work with numerical data stored in NumPy arrays. A vector of n observations is represented as:

$$x = (x_1, x_2, \ldots, x_n).$$

Code Example

Basic Operations with NumPy

```
import numpy as np

# Create an array
x = np.array([1.0, 2.0, 3.0, 4.0])

# Compute statistics
mean_x = np.mean(x)
var_x  = np.var(x)

# Vector operations
y = x * 2 + 1
print("Mean:", mean_x, "Variance:", var_x)
print("Transformed vector:", y)
```

Pandas

Pandas is the standard tool for working with tabular data. It provides the `DataFrame` structure, convenient for:

- loading datasets,
- inspecting data,
- handling missing values,
- feature selection.

Code Example

Pandas DataFrame Example

```
import pandas as pd

# Create a simple DataFrame
df = pd.DataFrame({
    "size": [50, 60, 80],
    "rooms": [2, 3, 4],
    "price": [150, 180, 240]
})

print(df.head())           # Inspect top rows
print(df.describe())       # Summary statistics
```

2.4.2 Visualization Tools

Visualizing data helps us understand patterns, detect outliers, and form hypotheses. The most common choice is `matplotlib`, often used together with `seaborn` for higher-level statistical plots.

Code Example

Basic Matplotlib Plot

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

x = np.linspace(0, 2*np.pi, 100)
y = np.sin(x)

plt.plot(x, y)
plt.title("Sine Wave")
plt.xlabel("x")
plt.ylabel("sin(x)")
plt.show()
```

2.4.3 scikit-learn: The Standard ML Toolkit

`scikit-learn` provides:

- implementations of classical machine learning algorithms,
- tools for preprocessing,
- model evaluation functions,
- utilities for dataset splitting and cross-validation.

The basic workflow for any supervised learning model in scikit-learn is:

1. Load or prepare the dataset.
2. Split into training and test sets.
3. Create a model object.
4. Fit the model on the training data.
5. Predict on unseen data.
6. Evaluate performance.

Below is a complete example following this workflow.

Code Example

Linear Regression on the California Housing Dataset

```python
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.datasets import fetch_california_housing

# 1. Load the California housing dataset
data = fetch_california_housing()
X = data.data        # features (2D array)
y = data.target      # target (1D array)

# 2. Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# 3. Create the model
model = LinearRegression()

# 4. Train the model
model.fit(X_train, y_train)

# 5. Predict on the test set
y_pred = model.predict(X_test)

# 6. Evaluate the model using MSE (mean squared error)
mse = mean_squared_error(y_test, y_pred)
print("MSE:", mse)
```

This example demonstrates the fundamental structure that will be reused for regression, classification, decision trees, ensemble methods, and deep learning models.

2.4.4 Linear and Logistic Regression in Python

In the previous chapter (see chapter 1), we developed the statistical foundations of linear and logistic regression. We now complement the theory with practical Python implementations using `scikit-learn`. The goal is to understand how these models are trained in practice, how predictions are produced, and how model performance is evaluated.

Linear Regression

Linear regression models a continuous target variable by fitting a linear relationship between a feature matrix X and an output vector y. The model is trained by minimizing the mean squared error (MSE), exactly as derived in the Least Squares solution.

Code Example

Linear Regression on a Simple Synthetic Dataset (Pedagogical Example)

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# ------------------------------------------------------------
# 1. Create a simple synthetic dataset
# ------------------------------------------------------------
# We assume the true relationship is:
#     y = 3*x + 5 + noise
# where noise ~ Normal(0, 1).

np.random.seed(0)
X = 2 * np.random.rand(100, 1)          # 100 samples, 1 feature
y = 3 * X[:, 0] + 5 + np.random.randn(100)  # true line + noise

# ------------------------------------------------------------
# 2. Train-test split
# ------------------------------------------------------------
# test_size=0.2 means:
# - 80 samples go to training (80%)
# - 20 samples go to testing (20%)
#
# random_state=42 ensures reproducible shuffling of the data.

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# ------------------------------------------------------------
# 3. Create the linear regression model
# ------------------------------------------------------------
# LinearRegression() fits:
#   y_hat = w*x + b
# to minimize the Mean Squared Error.

model = LinearRegression()

# ------------------------------------------------------------
# 4. Train the model
# ------------------------------------------------------------
# The model learns w (slope) and b (intercept)
# from the training data by solving the least-squares problem.

model.fit(X_train, y_train)
```

```
# Inspect learned parameters
print("Estimated slope (w):     ", model.coef_[0])
print("Estimated intercept (b): ", model.intercept_)

# ------------------------------------------------------------
# 5. Predict on the test set
# ------------------------------------------------------------
y_pred = model.predict(X_test)

# ------------------------------------------------------------
# 6. Compute the test MSE
# ------------------------------------------------------------
mse = mean_squared_error(y_test, y_pred)
print("Test MSE:", mse)
```

The workflow follows the standard pattern of scikit-learn: loading the dataset, splitting it, fitting the model, making predictions, and computing the MSE.

Logistic Regression

Logistic regression is used for binary classification. It models the probability of the positive class using the sigmoid function and is trained by maximizing the log-likelihood (equivalently, minimizing cross-entropy loss).

Code Example

Logistic Regression (scikit-learn)

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

# 1. Load dataset (binary classification)
data = load_breast_cancer()
X = data.data       # features
y = data.target     # labels (0 or 1)

# 2. Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=47
)

# 3. Build a pipeline: Standardize features -> Logistic Regression
# StandardScaler:
#   - subtracts the mean of each feature
```

```
#    - divides by the standard deviation
#    => each feature has mean 0 and variance 1
#
# LogisticRegression:
#    - now runs on scaled data
#    - we allow more iterations (max_iter=1000)

model = make_pipeline(
    StandardScaler(),
    LogisticRegression(max_iter=1000)
)

# 4. Train (fit) the model
model.fit(X_train, y_train)

# 5. Predict labels on the test set
y_pred = model.predict(X_test)

# 6. Evaluate with accuracy
acc = accuracy_score(y_test, y_pred)
print("Accuracy:", acc)
```

This implementation matches the statistical model described earlier in the notes: the model learns weights w so that $\sigma(w^\top x)$ approximates $P(y = 1 \mid x)$.

Comparison of Linear vs Logistic Regression

- **Linear Regression** predicts real-valued outputs using a linear model and the mean squared error loss. It is suitable for regression tasks such as price prediction or forecasting.
- **Logistic Regression** predicts probabilities using the sigmoid function and is trained using cross-entropy loss. It is suitable for binary classification tasks such as medical diagnosis or spam detection.
- Both models fit into the Empirical Risk Minimization framework described earlier:

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} L(y_i, f_\theta(x_i)).$$

- Both appear as special cases of Maximum Likelihood Estimation (Linear regression under Gaussian noise, logistic regression under a Bernoulli model).

3

Machine Learning Algorithms

In the previous chapters we developed the statistical foundations and learned how to use Python to implement linear and logistic regression. We now move to a powerful family of models that are widely used in practice:

- Decision Trees
- Random Forests
- Gradient Boosting
- XGBoost (including XGBRegressor)

All these methods are based on **decision trees**, but they use them in different ways. Understanding the tree first will make the rest much easier.

High-Level Picture

Key Idea

Scheme: How the algorithms relate

- **Decision Tree** One tree that splits the data into regions and makes a prediction in each region.
- **Random Forest** Many trees, each trained on a random subset of the data and features. Final prediction = average (regression) or majority vote (classification).

$$\text{Random Forest}(x) = \frac{1}{T}\sum_{t=1}^{T} f_t(x) \tag{3.1}$$

  where

  - x: a new input we want to predict
  - T: number of tree forest
  - f_t: the prediction of tree$t$ and output x.

- **Gradient Boosting** Trees are added one after another, each new tree tries to correct the errors (residuals) of the previous trees.

$$F_M(x) = F_{M-1}(x) + \nu \cdot h_M(x) \tag{3.2}$$

where:

- x: a new input we want to predict.
- $F_M(x)$: the boosted model after M trees.
- $F_{M-1}(x)$: the model built so far, using the first $M-1$ trees.
- $h_M(x)$: the prediction of the new tree M (a small correction).
- ν: the **learning rate**, a small number (e.g. 0.1) that controls how much the new tree influences the model.

- **XGBoost** A highly optimized implementation of gradient boosting with regularization, second–order gradients, and many engineering improvements. This is why it is so strong in practice.

Sum up:

Algorithm	Main Idea	Key Effect
Decision Tree	Single tree	Flexible but unstable
Random Forest	Many trees in parallel (bagging)	Reduces variance
Gradient Boosting	Trees in sequence (boosting)	Reduces bias
XGBoost	Optimized gradient boosting	High accuracy

In the following sections we study each algorithm in detail, both conceptually and with Python code.

3.1 Decision Trees

A decision tree is a model that makes predictions by asking a sequence of simple questions about the input features. Each question splits the data into two parts, and this splitting continues until we reach a leaf.

Intuition

Think of a flowchart:

$$\texttt{if rooms < 3} \rightarrow \text{go left } \texttt{else} \rightarrow \text{go right}$$

At the end of the path, the tree outputs:

- a number (average of y in that leaf) for regression, or
- a class (majority class in that leaf) for classification.

Thus, a decision tree divides the feature space into rectangular regions and assigns a simple prediction to each region.

Formal View

A regression tree predicts continuous values by recursively splitting the input space into regions that are as "pure" (homogeneous) as possible. The learning process follows a well-defined mathematical procedure.

At each node of the tree:

- We have a subset of the training data (x_i, y_i) that has reached this node.
- We consider binary splits of the form

$$x_j \leq t \quad \text{vs.} \quad x_j > t,$$

  where x_j is one feature and t is a threshold.
- For every candidate pair (j,t), we compute how much the **impurity** would decrease if we split the node according to that rule.

Impurity measure for regression. For regression, impurity is typically measured using the Mean Squared Error (MSE) within the node:

$$\text{MSE}(\text{node}) = \frac{1}{n_{\text{node}}} \sum_{i \in \text{node}} \left(y_i - \bar{y}_{\text{node}}\right)^2,$$

where:

- n_{node} is the number of samples in the node,
- $\bar{y}_{\text{node}}$ is the average target value of the samples in the node,
- y_i is the true target of sample i.

Interpretation: A node has low MSE when all y_i values are close to their mean. This means the node is *pure*: all examples inside it behave similarly.

Choosing the best split. For each candidate split (j,t), the algorithm computes the impurity after splitting:

$$\text{Impurity\_after} = \frac{n_L}{n}\,\text{MSE}(L) + \frac{n_R}{n}\,\text{MSE}(R),$$

where:

- L and R are the left and right child nodes,
- $n = n_L + n_R$ is the number of samples in the parent node.

The **impurity reduction** (also called the "gain") is:

$$\Delta = \mathrm{MSE}(\text{parent}) - \left[\frac{n_L}{n}\,\mathrm{MSE}(L) + \frac{n_R}{n}\,\mathrm{MSE}(R)\right].$$

The algorithm chooses the split (j,t) that maximizes Δ. A large Δ means the split creates two children that are more homogeneous (pure) than the parent.

Recursive growth. After choosing the best split:

1. Assign samples to the left or right child depending on the split.
2. Repeat the same procedure recursively on each child node.

The process stops when a stopping condition is met, such as:

- the node contains too few samples,
- the impurity is already very low,
- the maximum depth has been reached.

Each terminal node (leaf) stores a prediction equal to the **mean** of the y_i values in that leaf.

Note

A regression tree searches for feature thresholds that create child nodes where the target values are as similar as possible. The objective is always the same:

$$\text{make each region pure} \quad \Longrightarrow \quad \text{reduce MSE as much as possible.}$$

Hyperparameters (What controls a tree?)

Common hyperparameters:

- `max_depth`: maximum depth of the tree.
- `min_samples_split`: minimum samples to split a node.
- `min_samples_leaf`: minimum samples in a leaf.

Key Idea

Shallow tree (small depth) $\Rightarrow$ high bias, low variance.
Deep tree (large depth) $\Rightarrow$ low bias, high variance (easy to overfit).

Advantages and Disadvantages

- **Advantages**
  - Easy to visualize and interpret.
  - Can capture nonlinear relationships and interactions.
  - No need for feature scaling.
- **Disadvantages**
  - Very sensitive to small changes in the data.
  - Easily overfits if grown too deep.
  - Usually not as accurate as ensemble methods.

Python Example: Decision Tree Regression

Code Example

Decision Tree Regression Example

```python
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error

# Load dataset
data = fetch_california_housing()
X, y = data.data, data.target

# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Train model
tree = DecisionTreeRegressor(max_depth=5)
# We  decided to specify only that hyperparameter.
# All other hyperparameters take their default values.
tree.fit(X_train, y_train)

# Predict
y_pred = tree.predict(X_test)

# Evaluate
mse = mean_squared_error(y_test, y_pred)
print("Decision Tree MSE:", mse)
```

Note

Markov Chain Monte Carlo (MCMC): A Bayesian Inference Tool

Although not a machine learning model, Markov Chain Monte Carlo (MCMC) is often mentioned in statistical learning contexts. It belongs to **Bayesian statistics**, not predictive modelling. Its purpose is to approximate *probability distributions*, not to make predictions.

What MCMC Does

MCMC is used when we want to compute a posterior distribution

$$P(\theta \mid x) \propto P(x \mid \theta) P(\theta),$$

but the denominator

$$P(x) = \int P(x \mid \theta) P(\theta) \, d\theta$$

is too difficult to compute analytically.

Instead of solving the integral, MCMC constructs a **Markov chain** whose long-run behavior approximates the posterior distribution. After many iterations, the samples

$$\theta^{(1)}, \, \theta^{(2)}, \, \ldots, \, \theta^{(M)}$$

act as draws from $P(\theta \mid x)$.

This allows us to compute expectations, medians, quantiles, and credible intervals.

Example: Posterior Over a Poisson Rate λ

Suppose data follows a Poisson model:

$$x_i \sim \text{Poisson}(\lambda).$$

We choose a Gamma prior:

$$\lambda \sim \text{Gamma}(\alpha, \beta).$$

The posterior is:

$$P(\lambda \mid x) \propto \lambda^{\Sigma_i x_i + \alpha - 1} \exp[-\lambda (n + \beta)] \, .$$

Rather than computing this posterior analytically, MCMC generates many samples $\lambda^{(1)}, \lambda^{(2)}, \ldots$ that approximate its shape. From these, we estimate:

$$\mathbb{E}[\lambda \mid x] \approx \frac{1}{M} \sum_{m=1}^{M} \lambda^{(m)}.$$

Conceptual Understanding of MCMC

- Start at an initial guess $\theta^{(0)}$.
- Propose a nearby value.
- Accept or reject based on how plausible it is under the posterior.

- Repeat to explore the distribution.

Common algorithms: Metropolis–Hastings, Gibbs Sampling.

How MCMC Differs from Decision Trees

- **Decision Tree**:
  - A *supervised learning model*.
  - Learns a function $f(x)$ to predict y.
  - Builds splitting rules to reduce impurity (MSE or Gini).
  - Produces predictions.
- **MCMC**:
  - A *Bayesian sampling method*.
  - Approximates a posterior distribution $P(\theta \mid x)$.
  - Does not create a predictive model.
  - Produces samples representing uncertainty in parameters.

Key Difference

Decision Trees predict outcomes. MCMC quantifies uncertainty in parameters. One is a model; the other is a sampling algorithm.

3.2 Random Forest

A random forest is an **ensemble** of decision trees. Instead of relying on one tree, we train many trees and average their predictions.

Intuition

The idea is similar to asking many different experts and averaging their answers.

Key Idea

A single tree has high variance. A forest of many trees, each slightly different, has lower variance.

How do we make the trees different?

- **Bootstrap sampling (bagging)**: each tree is trained on a different random sample of the training data, sampled *with replacement*.
- **Random feature selection**: at each split, the tree considers a random subset of features instead of all of them.

Prediction Rule

For regression:

$$\hat{y}(x) = \frac{1}{T}\sum_{t=1}^{T} f_t(x),$$

where f_t is the prediction of tree t.

For classification:

$$\hat{y}(x) = \text{majority vote of the trees.}$$

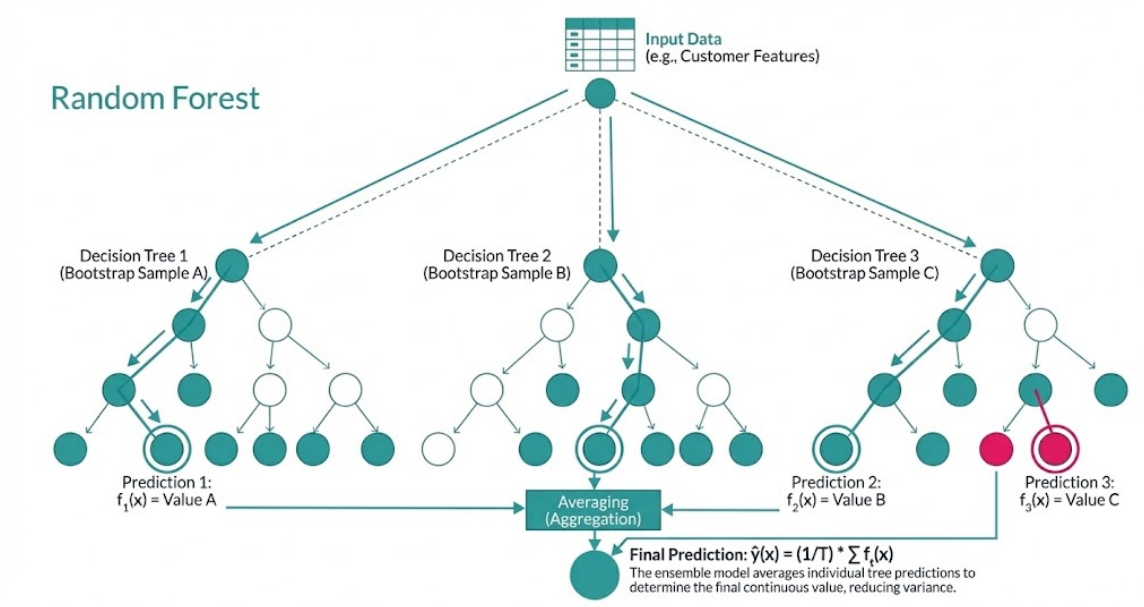

Figure 3.1: Diagram illustrating the Random Forest algorithm and majority voting process.

Effect on Bias and Variance

- Each tree is a high-variance, low-bias model.
- Averaging many trees reduces variance while keeping bias roughly the same.
- This usually improves generalization significantly.

Important Hyperparameters

- `n_estimators`: Controls the **number of trees** in the forest. This affects the *ensemble level*: more trees usually reduce variance and improve stability.
- `max_depth`: Limits the **maximum depth of each individual tree**. Acts at the *tree level*. Smaller depth $\rightarrow$ simpler trees $\rightarrow$ less overfitting.

- `max_features`: Controls how many features are considered when searching for the best split *at each node*. Acts at the *split level*. Adding randomness here reduces correlation between trees.
- `min_samples_leaf`: Sets the minimum number of samples required to form a leaf. Acts at the *leaf level*. Larger values force leaves to contain more data → smoother predictions.

Python Example: Random Forest Regression

Code Example

Random Forest Regression Example

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Load dataset
data = fetch_california_housing()
X, y = data.data, data.target

# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Train model
forest = RandomForestRegressor(
    n_estimators=200,
    max_depth=None,
    random_state=42
)
forest.fit(X_train, y_train)

#forest = RandomForestRegressor(
#    n_estimators=300,          # number of trees
#    max_depth=12,              # maximum depth of each tree
#    max_features="sqrt",       # number of features tested at each split
#    min_samples_leaf=3,        # minimum samples per leaf
#    min_samples_split=4,       # minimum samples required to #split a
   node
#    random_state=42            # reproducibility
#)

# Predict
y_pred = forest.predict(X_test)

# Evaluate
mse = mean_squared_error(y_test, y_pred)
```

```
print("Random Forest MSE:", mse)
```

Code Example

Random Forest Regression on Synthetic $\sin(x)$ Data (with Plot)

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

# ---------------------------------------------------------
# 1. Create synthetic data for y = sin(x) + noise
# ---------------------------------------------------------
np.random.seed(0)

n_samples = 200
X = np.linspace(0, 2*np.pi, n_samples).reshape(-1, 1)
y_true_function = np.sin(X[:, 0])
noise = 0.1 * np.random.randn(n_samples)
y = y_true_function + noise

# ---------------------------------------------------------
# 2. Train-test split
# ---------------------------------------------------------
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# ---------------------------------------------------------
# 3. Random Forest model
# ---------------------------------------------------------
forest = RandomForestRegressor(
    n_estimators=200,
    max_depth=8,
    random_state=42
)

# ---------------------------------------------------------
# 4. Train the model
# ---------------------------------------------------------
forest.fit(X_train, y_train)

# ---------------------------------------------------------
# 5. Predict on test set
# ---------------------------------------------------------
```

```
y_pred = forest.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print("Test MSE (Random Forest on sin(x)):", mse)

# ------------------------------------------------------------
# 6. Plot: true function, noisy data, and model predictions
# ------------------------------------------------------------
# Sort for nice plotting
sort_idx = np.argsort(X_test[:, 0])
X_test_sorted = X_test[sort_idx]
y_test_sorted = y_test[sort_idx]
y_pred_sorted = y_pred[sort_idx]

# High-resolution grid for smooth model curve
X_grid = np.linspace(0, 2*np.pi, 1000).reshape(-1, 1)
y_grid_pred = forest.predict(X_grid)

plt.figure(figsize=(10, 5))

# True function (smooth)
plt.plot(X[:, 0], y_true_function, label="True sin(x)", color="black",
    linewidth=2)

# Noisy training data
plt.scatter(X_train[:, 0], y_train, label="Noisy training data", alpha
    =0.5)

# Random Forest smooth prediction
plt.plot(X_grid[:, 0], y_grid_pred, label="Random Forest prediction",
    color="red", linewidth=2)

plt.legend()
plt.title("Random Forest Regression on Noisy $\sin(x)$ Data")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.show()

# ------------------------------------------------------------
# 7. Print a small table: true vs predicted
# ------------------------------------------------------------
print("\nFirst 10 test points: true vs predicted")
for x_val, y_t, y_p in zip(X_test_sorted[:10, 0],
                           y_test_sorted[:10],
                           y_pred_sorted[:10]):
    print(f"x = {x_val:.2f},  y_true = {y_t:.3f},  y_pred = {y_p:.3f}")
```

3.3 Gradient Boosting

Random forests build trees in *parallel*. Gradient boosting builds trees *sequentially*: each new tree corrects the errors of the previous model.

Intuition

We start with a simple model $F_0(x)$ (for example a constant). At each step $m = 1, 2, \ldots, M$ we add a new tree $h_m(x)$:

$$F_m(x) = F_{m-1}(x) + \nu\, h_m(x),$$

where ν is the learning rate.

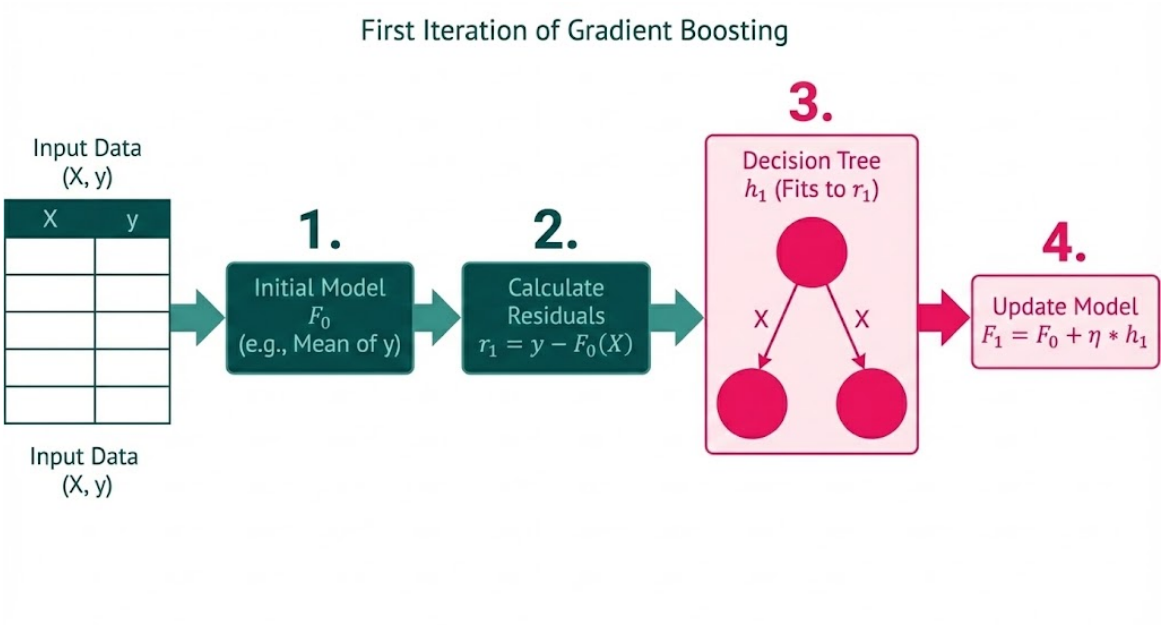

Figure 3.2: Diagram illustrating the Gradient Boosting.

What does $h_m(x)$ learn? It is trained to fit the **residuals** of the previous model:

$$r_i^{(m)} = y_i - F_{m-1}(x_i).$$

So each tree focuses on what the current model is doing badly.

Key Idea

Random forest: many independent trees averaged. Gradient boosting: a sequence of trees, each correcting the previous ones.

Optimization View

More generally, gradient boosting fits trees to the *negative gradient* of the loss function with respect to the current predictions. This is why it is called **gradient** boosting.

To see this mathematically, suppose our model at iteration m is $F_{m-1}(x)$, and we want to minimize the empirical risk

$$R(F) = \frac{1}{n}\sum_{i=1}^{n} L\big(y_i, F(x_i)\big).$$

The gradient of the loss with respect to the model's prediction at point x_i is:

$$g_i^{(m)} = \left.\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right|_{F=F_{m-1}}.$$

Gradient boosting uses the *negative gradient* as a pseudo-target:

$$r_i^{(m)} = -g_i^{(m)}.$$

Note

The negative gradient $r_i^{(m)}$ gives the direction in which the loss decreases the fastest. For squared loss,

$$L(y_i, F) = (y_i - F)^2, \quad \Longrightarrow \quad r_i^{(m)} = y_i - F_{m-1}(x_i),$$

so the negative gradient is exactly the *residual*.

The new tree $h_m(x)$ is trained to approximate these pseudo-targets:

$$h_m(x) \approx r^{(m)}.$$

The model is then updated by taking a small step in that direction:

$$F_m(x) = F_{m-1}(x) + \nu\, h_m(x),$$

where $0 < \nu \leq 1$ is the learning rate.

Key Idea

Gradient boosting is gradient descent in *function space*: each tree h_m points in the direction that most reduces the loss, and the learning rate ν controls how big that step is.

Hyperparameters

- `n_estimators`: number of boosting iterations (trees).
- `learning_rate` ν: step size; smaller values need more trees.

- `max_depth`: depth of each individual tree (usually small, e.g. 3–5).

Python Example: Gradient Boosting Regression

Code Example

Gradient Boosting Regression

```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

data = fetch_california_housing()
X, y = data.data, data.target

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

gbr = GradientBoostingRegressor(
    n_estimators=300,
    learning_rate=0.05,
    max_depth=3
)
gbr.fit(X_train, y_train)

y_pred = gbr.predict(X_test)
print("GBR MSE:", mean_squared_error(y_test, y_pred))
```

Code Example

Gradient Boosting Regression on Synthetic $\sin(x)$ Data (with Plot)

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error

# ------------------------------------------------------------
# 1. Create synthetic data for y = sin(x) + noise
#    (same setup as for the Random Forest example)
# ------------------------------------------------------------
np.random.seed(0)

n_samples = 200
X = np.linspace(0, 2*np.pi, n_samples).reshape(-1, 1)
```

```
y_true_function = np.sin(X[:, 0])
noise = 0.1 * np.random.randn(n_samples)
y = y_true_function + noise

# ------------------------------------------------------------
# 2. Train-test split
# ------------------------------------------------------------
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# ------------------------------------------------------------
# 3. Gradient Boosting model
# ------------------------------------------------------------
# n_estimators  : number of boosting stages (trees)
# learning_rate : step size for each tree's contribution
# max_depth     : depth of individual regression trees
# random_state  : reproducibility

gboost = GradientBoostingRegressor(
    n_estimators=300,
    learning_rate=0.05,
    max_depth=3,
    random_state=42
)

# ------------------------------------------------------------
# 4. Train the model
# ------------------------------------------------------------
gboost.fit(X_train, y_train)

# ------------------------------------------------------------
# 5. Predict on test set and compute MSE
# ------------------------------------------------------------
y_pred = gboost.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print("Test MSE (Gradient Boosting on sin(x)):", mse)

# ------------------------------------------------------------
# 6. Plot: true function, noisy data, and GB prediction
# ------------------------------------------------------------
# Sort for nice plotting
sort_idx = np.argsort(X_test[:, 0])
X_test_sorted = X_test[sort_idx]
y_test_sorted = y_test[sort_idx]
y_pred_sorted = y_pred[sort_idx]

# High-resolution grid for smooth model curve
```

```
X_grid = np.linspace(0, 2*np.pi, 1000).reshape(-1, 1)
y_grid_pred = gboost.predict(X_grid)

plt.figure(figsize=(10, 5))

# True function (smooth)
plt.plot(X[:, 0], y_true_function, label="True sin(x)", color="black",
    linewidth=2)

# Noisy training data
plt.scatter(X_train[:, 0], y_train, label="Noisy training data", alpha
    =0.5)

# Gradient Boosting prediction
plt.plot(X_grid[:, 0], y_grid_pred, label="Gradient Boosting prediction",
         color="green", linewidth=2)

plt.legend()
plt.title("Gradient Boosting Regression on Noisy $\sin(x)$ Data")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.show()

# ------------------------------------------------------------
# 7. Print a small table: true vs predicted
# ------------------------------------------------------------
print("\nFirst 10 test points: true vs predicted (Gradient Boosting)")
for x_val, y_t, y_p in zip(X_test_sorted[:10, 0],
                           y_test_sorted[:10],
                           y_pred_sorted[:10]):
    print(f"x = {x_val:.2f},  y_true = {y_t:.3f},  y_pred = {y_p:.3f}")
```

3.4 XGBoost: Extreme Gradient Boosting

Now we can finally answer: **What is XGBoost?**

XGBoost is a specific, highly optimized implementation of gradient boosting for decision trees. It uses the same basic idea (add trees sequentially to correct errors), but introduces:

- strong regularization of trees,
- second-order (Hessian) information,
- efficient handling of sparse inputs,
- parallelization and system optimizations.

Because of these improvements, XGBoost tends to be:

- faster to train,
- more accurate,
- better at controlling overfitting,

than a naive gradient boosting implementation.

Objective Function

The model at iteration t is:

$$\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(x_i),$$

where each f_k is a decision tree.

XGBoost minimizes the regularized objective:

$$\mathscr{L}^{(t)} = \sum_{i=1}^{n} L\big(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\big) + \Omega(f_t),$$

with regularization term:

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2,$$

where:

- T = number of leaves in the tree,
- w_j = weight (value) assigned to leaf j,
- γ and λ are regularization hyperparameters.

This penalizes both the *size* of the tree (number of leaves) and the *magnitude* of leaf weights, which helps prevent overfitting.

Second-Order Approximation

To efficiently choose splits and leaf values, XGBoost uses a second-order Taylor expansion of the loss around the current predictions:

$$L(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) \approx L(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2,$$

where:

- $g_i = \partial_{\hat{y}} L(y_i, \hat{y}_i^{(t-1)})$ (gradient),
- $h_i = \partial^2_{\hat{y}} L(y_i, \hat{y}_i^{(t-1)})$ (Hessian).

This second-order information makes tree construction more accurate and efficient.

Key Hyperparameters (What you tune in practice)

- `n_estimators`: number of boosting rounds (trees).
- `learning_rate`: shrinkage factor for each tree contribution.
- `max_depth`: maximum depth of each tree.
- `subsample`: fraction of training data used per tree.
- `colsample_bytree`: fraction of features used per tree.
- `reg_lambda`, `reg_alpha`: L2 and L1 regularization on leaf weights.

XGBoost Classification Example

Code Example

XGBoost Classifier

```
from xgboost import XGBClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

data = load_breast_cancer()
X, y = data.data, data.target

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

model = XGBClassifier(
    n_estimators=300,
    learning_rate=0.05,
    max_depth=4,
    subsample=0.8,
    colsample_bytree=0.8,
    eval_metric="logloss"
)

model.fit(X_train, y_train)
y_pred = model.predict(X_test)

print("XGBoost Accuracy:", accuracy_score(y_test, y_pred))
```

XGBoost Regression Example

Code Example

XGBoost Regressor

```
from xgboost import XGBRegressor
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

data = fetch_california_housing()
X, y = data.data, data.target

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

model = XGBRegressor(
    n_estimators=400,
    learning_rate=0.05,
    max_depth=5,
    subsample=0.8,
    colsample_bytree=0.8,
    eval_metric="rmse"
)

model.fit(X_train, y_train)
y_pred = model.predict(X_test)

print("XGBRegressor MSE:", mean_squared_error(y_test, y_pred))
```

Code Example

XGBoost Regression on Synthetic $\sin(x)$ Data (with Plot)

```

import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

from xgboost import XGBRegressor   # pip install xgboost

# ------------------------------------------------------------
# 1. Create synthetic data for y = sin(x) + noise
```

```
#    (same setup as RF and Gradient Boosting examples)
# ------------------------------------------------------------
np.random.seed(0)

n_samples = 200
X = np.linspace(0, 2*np.pi, n_samples).reshape(-1, 1)
y_true_function = np.sin(X[:, 0])
noise = 0.1 * np.random.randn(n_samples)
y = y_true_function + noise

# ------------------------------------------------------------
# 2. Train-test split
# ------------------------------------------------------------
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# ------------------------------------------------------------
# 3. XGBoost model
# ------------------------------------------------------------
# n_estimators      : number of boosting trees
# learning_rate     : step size (eta)
# max_depth         : depth of individual trees
# subsample         : fraction of rows used per tree (stochasticity)
# colsample_bytree  : fraction of features used per tree
# reg_lambda        : L2 regularization
# objective         : regression with squared loss

xgb_model = XGBRegressor(
    n_estimators=400,
    learning_rate=0.05,
    max_depth=3,
    subsample=0.9,
    colsample_bytree=0.9,
    reg_lambda=1.0,
    objective="reg:squarederror",
    random_state=42
)

# ------------------------------------------------------------
# 4. Train the model
# ------------------------------------------------------------
xgb_model.fit(X_train, y_train)

# ------------------------------------------------------------
# 5. Predict on test set and compute MSE
# ------------------------------------------------------------
y_pred = xgb_model.predict(X_test)
```

```
mse = mean_squared_error(y_test, y_pred)
print("Test MSE (XGBoost on sin(x)):", mse)

# ------------------------------------------------------------
# 6. Plot: true function, noisy data, and XGBoost prediction
# ------------------------------------------------------------
# Sort test data for nicer display
sort_idx = np.argsort(X_test[:, 0])
X_test_sorted = X_test[sort_idx]
y_test_sorted = y_test[sort_idx]
y_pred_sorted = y_pred[sort_idx]

# High-resolution grid for smooth model curve
X_grid = np.linspace(0, 2*np.pi, 1000).reshape(-1, 1)
y_grid_pred = xgb_model.predict(X_grid)

plt.figure(figsize=(10, 5))

# True function
plt.plot(X[:, 0], y_true_function, label="True sin(x)",
         color="black", linewidth=2)

# Noisy training data
plt.scatter(X_train[:, 0], y_train, label="Noisy training data",
            alpha=0.5)

# XGBoost prediction
plt.plot(X_grid[:, 0], y_grid_pred, label="XGBoost prediction",
         color="purple", linewidth=2)

plt.legend()
plt.title("XGBoost Regression on Noisy $\sin(x)$ Data")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.show()

# ------------------------------------------------------------
# 7. Print a small table: true vs predicted
# ------------------------------------------------------------
print("\nFirst 10 test points: true vs predicted (XGBoost)")
for x_val, y_t, y_p in zip(X_test_sorted[:10, 0],
                           y_test_sorted[:10],
                           y_pred_sorted[:10]):
    print(f"x = {x_val:.2f},  y_true = {y_t:.3f},  y_pred = {y_p:.3f}")
```

3.5 Summary: Recognizing and Distinguishing the Algorithms

Example-Recap

How to quickly identify each model

- **Decision Tree**
  - Single tree, flowchart-like structure.
  - Splits the data based on feature thresholds.
  - Very interpretable but can overfit easily.
- **Random Forest**
  - Many trees trained on bootstrap samples and random subsets of features.
  - Predictions are averaged (regression) or voted (classification).
  - Reduces variance and is robust; default strong baseline.
- **Gradient Boosting**
  - Trees are added one after another.
  - Each new tree focuses on residuals / gradients of the previous model.
  - Can fit complex patterns with relatively shallow trees.
- **XGBoost / XGBRegressor**
  - A particular implementation of tree-based gradient boosting.
  - Uses regularization, second-order gradients, and many optimizations.
  - Often achieves state-of-the-art performance on tabular datasets.

Rule of thumb:

- Need interpretability and a quick idea? Try a single *decision tree*.
- Need a strong, robust model with little tuning? Use a *random forest*.
- Need maximum performance and are willing to tune hyperparameters? Use *gradient boosting* or *XGBoost*.

4

Neural Networks & Deep Learning

Neural networks are among the most powerful and widely used models in modern machine learning. They are capable of learning highly nonlinear relationships, complex patterns in images, speech, text, and even reinforcement-learning tasks.

This chapter presents the essential ideas behind neural networks and deep learning in a clear, intuitive, and mathematically grounded way.

What Makes Neural Networks Special?

Neural networks do not rely on manually–designed features. Instead, they learn representations directly from data.

Key Idea

Neural networks learn *hierarchies of features*: simple patterns at shallow layers, more complex patterns at deeper layers.

They are versatile enough to solve:

- regression problems,
- classification problems,
- image analysis (CNNs),
- sequential modelling (RNNs, LSTMs, Transformers),
- reinforcement learning,
- generative modelling.

In this chapter we focus on the foundations needed to understand all these models.

4.1 The Perceptron: The Simplest Neural Unit

The basic building block of a neural network is the **neuron** (or node). The simplest neuron is the classical **perceptron**.

Mathematical Form

Given an input vector $x \in \mathbb{R}^d$:

$$z = w^\top x + b$$

and an activation:

$$\hat{y} = \sigma(z)$$

where:

- w = weights,
- b = bias,
- $\sigma(\cdot)$ = activation function.

Key Idea

Intuition: What is a neural network?
A neural network is best understood as a *chain of transformations*. It takes the raw input x and gradually transforms it into more useful representations:

$$x \longrightarrow h^{(1)} \longrightarrow h^{(2)} \longrightarrow h^{(3)} \longrightarrow \hat{y}.$$

Each layer computes

$$h^{(1)} = \sigma\left(W^{(1)}x + b^{(1)}\right), \qquad h^{(2)} = \sigma\left(W^{(2)}h^{(1)} + b^{(2)}\right),$$

and so on.

What does this mean intuitively?

- The input x is the raw data (pixels, temperature, salary, age, etc.).
- The first layer $h^{(1)}$ becomes a new description of the data:

$$h^{(1)} = (\text{“edge on the left?”}, \text{“is it big?”}, \text{“is it bright?”}, \dots)$$

- The second layer $h^{(2)}$ detects more complex features built from $h^{(1)}$:

$$h^{(2)} = (\text{“is there a vertical shape?”}, \text{“does this look like an eye?”}, \dots)$$

- Deeper layers combine simpler patterns into abstract concepts:

$$\text{edges} \to \text{shapes} \to \text{parts} \to \text{full objects}.$$

Hierarchy of features:

Layer 1: simple patterns Layer 2: combinations Layer 3: abstract concepts

Each layer's output is a feature vector built on top of the previous one. This progressive "simple $\rightarrow$ complex" structure is called a *hierarchy of representations*, and it is the reason neural networks can model rich, nonlinear patterns in data.

Activation Functions

A neuron computes a weighted sum

$$z = w^\top x + b,$$

and then applies an *activation function* $\sigma(z)$.

$\sigma(z)$ introduces nonlinearity.

Without this nonlinearity, every layer of the network would be linear, and the entire neural network would collapse into a single linear model. Activation functions therefore give neural networks their expressive power.

Common activation functions:

- **Sigmoid:**
$$\sigma(z) = \frac{1}{1+e^{-z}}$$
Maps real numbers to $(0,1)$. Useful for probabilities and binary classification.
- **ReLU (Rectified Linear Unit):**
$$\sigma(z) = \max(0,z)$$
Simple and efficient. Keeps positive values, sets negative values to zero. Helps deep networks learn complex patterns without saturating.
- **Tanh:**
$$\sigma(z) = \tanh(z)$$
Outputs values in $(-1,1)$. Zero-centered and smoother than sigmoid.

Note

Intuition: Why do we need activation functions?
The linear part $w^\top x + b$ can only produce straight-line relationships. A stack of linear layers is still just linear.
Activation functions bend and shape these straight lines, allowing the network to learn curves, interactions, and complex patterns.

Linear layers learn combinations of inputs, activation functions let them learn structure.

ReLU detects whether a pattern is present ("is this feature active?"), sigmoid produces smooth probabilities, and tanh gives graded negative/positive activations.
Without $\sigma(\cdot)$, a neural network cannot learn anything nonlinear. With activation functions, it becomes a universal function approximator.

Note

• If we remove activation functions, a neural network collapses into a linear regression model. Nonlinear activations give neural networks their expressive power.

Why a network without activation is just linear regression
A neuron computes

$$h = \sigma(w^\top x + b).$$

If we remove the activation, i.e. $\sigma(z) = z$, it becomes

$$h = w^\top x + b,$$

a purely linear transformation.
A second layer without activation gives:

$$w_2^\top h + b_2 = w_2^\top (w_1^\top x + b_1) + b_2 = (Wx) + b,$$

which is still linear in x.
Thus, stacking any number of linear layers yields another linear function:

$$f(x) = W^* x + b^*,$$

exactly the form of linear regression.
Only nonlinear activations (ReLU, sigmoid, tanh) allow neural networks to represent nonlinear functions.

4.2 Multi–Layer Perceptron (MLP)

A **Multi–Layer Perceptron** is the simplest form of a neural network consisting of layers stacked one after another:

$$\text{input} \longrightarrow \text{hidden layers} \longrightarrow \text{output}.$$

Each layer takes the output of the previous layer, applies a linear transformation, and then a nonlinearity (activation). A **hidden layer** (or hidden sector) is a part of the neural network where intermediate computations take place: it receives the input from the previous layer, transforms it into a new feature representation, and passes it forward— but these intermediate values are never observed in the dataset, which is why they are called

"hidden."

Structure

Formally, for an input x:

$$h^{(1)} = \sigma\left(W^{(1)}x + b^{(1)}\right),$$
$$h^{(2)} = \sigma\left(W^{(2)}h^{(1)} + b^{(2)}\right),$$
$$\vdots$$
$$\hat{y} = W^{(L)}h^{(L-1)} + b^{(L)}.$$

```
Input → Layer 1 → Layer 2 → ...  → Output
```

Each additional layer increases the expressiveness of the model.

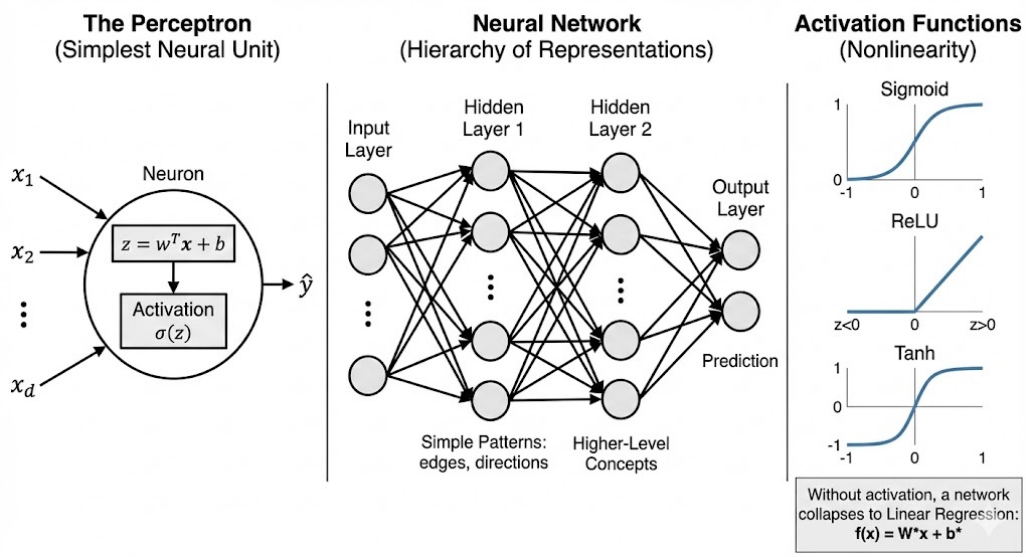

Figure 4.1: This composite diagram illustrates the three levels of a Deep Learning model: *(Left)* The architecture of a Multi-Layer Perceptron (MLP) with input (x), hidden (h), and output (y) layers. *(Top Right)* The internal mechanism of a single perceptron, computing the weighted sum $z = w^\top x + b$ followed by an activation. *(Bottom Right)* Common activation functions (σ)—Sigmoid, Tanh, and ReLU—that introduce the necessary nonlinearity into the network.

Key Idea

Intuition: What is a multi–layer network?

Think of an MLP as a *chain of transformations* applied to the input. Each hidden layer produces a new "version" of the data:

$$x \to h^{(1)} \to h^{(2)} \to h^{(3)} \to \hat{y}.$$

Each transformation has the form:

$$h^{(\ell)} = \sigma\left(W^{(\ell)}h^{(\ell-1)} + b^{(\ell)}\right).$$

- The input x is raw information (pixels, temperature, age, salary, etc.).
- The first hidden layer $h^{(1)}$ extracts **simple features** (edges, brightness, linear combinations, basic correlations).
- The second layer $h^{(2)}$ combines those simple features into **more complex patterns** (shapes, textures, interactions).
- Deeper layers form **abstract concepts** (object parts, objects, categories, high-level decisions).

This progressive "simple $\rightarrow$ complex" structure is what we call a **hierarchy of representations**. Each layer builds on the one before it. This is why deep networks can model extremely rich, nonlinear patterns.

Note

A neural network with one hidden layer and enough neurons can approximate *any continuous function*. This is the **Universal Approximation Theorem**.

4.3 Forward Pass and Loss Function

During training, the network performs:

1. **Forward pass:** compute predictions $\hat{y}$.
2. **Compute loss:** measure how wrong the predictions are.
3. **Backward pass:** compute gradients of the loss with respect to each parameter.
4. **Parameter update:** adjust weights to reduce the loss.

For regression, the loss is MSE:

$$L = \frac{1}{n}\sum(y_i - \hat{y}_i)^2.$$

For classification, the loss is cross–entropy:

$$L = -\sum y_i \log(\hat{p}_i).$$

4.4 Backpropagation: How Neural Networks Learn

Backpropagation is the algorithm that computes all gradients efficiently.

Chain Rule

Backprop uses the chain rule repeatedly:

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial h^{(l)}} \cdot \frac{\partial h^{(l)}}{\partial z^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial W^{(l)}}.$$

This allows gradients to flow from the output backward. Here is the meaning of each term:

- $\frac{\partial L}{\partial h^{(l)}}$ measures *how much the loss would change* if the output of layer l (the activations $h^{(l)}$) changed a little bit. This gradient comes from the layer above (layer $l+1$).
- $\frac{\partial h^{(l)}}{\partial z^{(l)}}$ is the derivative of the activation function $\sigma(z)$. It measures how sensitive the neuron output is to changes in its input:
$$h^{(l)} = \sigma(z^{(l)}).$$
- $\frac{\partial z^{(l)}}{\partial W^{(l)}}$ comes from the linear part
$$z^{(l)} = W^{(l)} h^{(l-1)} + b^{(l)}.$$
It tells us how much $z^{(l)}$ changes when we change the weights $W^{(l)}$. For fully connected layers:
$$\frac{\partial z^{(l)}}{\partial W^{(l)}} = h^{(l-1)}.$$

Putting these pieces together:

$$\frac{\partial L}{\partial W^{(l)}} = \underbrace{\frac{\partial L}{\partial h^{(l)}}}_{\text{error coming from next layer}} \cdot \underbrace{\sigma'(z^{(l)})}_{\text{activation derivative}} \cdot \underbrace{h^{(l-1)}}_{\text{input to the layer}}.$$

This is exactly the update signal used to adjust the weights.

Backpropagation consists of applying this formula layer by layer, from the output layer back toward the input.

> **Key Idea**
>
> Backpropagation = computationally efficient application of the chain rule.

4.5 Training with Gradient Descent

Once backpropagation has computed all required gradients, the parameters of the network are updated using **gradient descent**. Each parameter moves in the direction that most reduces the loss:

$$W \leftarrow W - \eta \frac{\partial L}{\partial W},$$

where η (eta) is the *learning rate*. This simple rule means:

- $\frac{\partial L}{\partial W}$ tells us how the loss changes if W changes.
- The update subtracts this quantity, meaning we move the parameter in the direction that reduces the loss.

- The learning rate η controls how big each step is.

In practice, many variations of gradient descent exist. They differ in how they compute the update direction and step size, but they all follow the same basic idea: use gradients to iteratively improve the parameters.

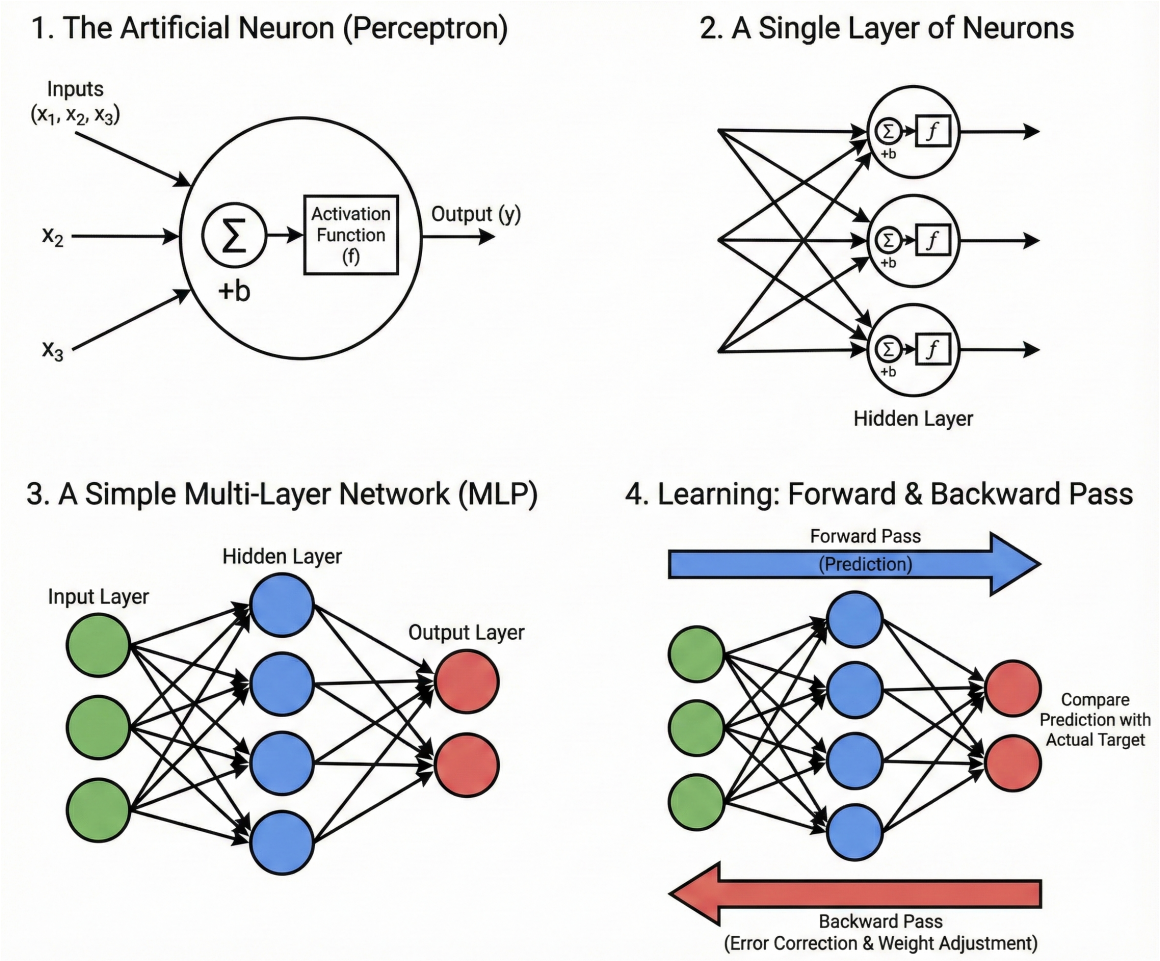

Figure 4.2: **From Neuron to Network: The Anatomy of Deep Learning.** Visualizing the hierarchy of a neural network: individual inputs (x) are processed by neurons using weights (w) and biases (b), then stacked into layers to form a complete Deep Learning architecture.

Common Optimizers

- **SGD (Stochastic Gradient Descent)** Updates parameters using a small batch (or even a single example). Faster per-step and introduces helpful randomness.
- **Momentum** Accumulates past gradients to smooth updates and accelerate learning.
- **RMSProp** Adapts the learning rate for each parameter based on recent gradient magnitudes.
- **Adam** Combines Momentum and RMSProp. One of the most widely used optimizers in deep learning.

Note

The learning rate is one of the most important hyperparameters. If it is too large, the updates overshoot and training diverges. If it is too small, training becomes extremely slow.

4.6 Why Deep Networks Work

The strength of deep networks comes from their ability to build a *hierarchy of representations*. Each layer transforms the data into a more useful and abstract form.

- **Shallow layers** detect simple, local patterns (edges, corners, brightness changes).
- **Intermediate layers** combine these into richer features (textures, contours, repeated patterns).
- **Deeper layers** assemble high-level concepts (shapes, object parts, or semantic relationships).
- **Final layers** recognize full objects or produce task-specific predictions.

This progressive "simple → structured → abstract" pipeline explains why depth matters: **each additional layer gives the network more expressive power**.

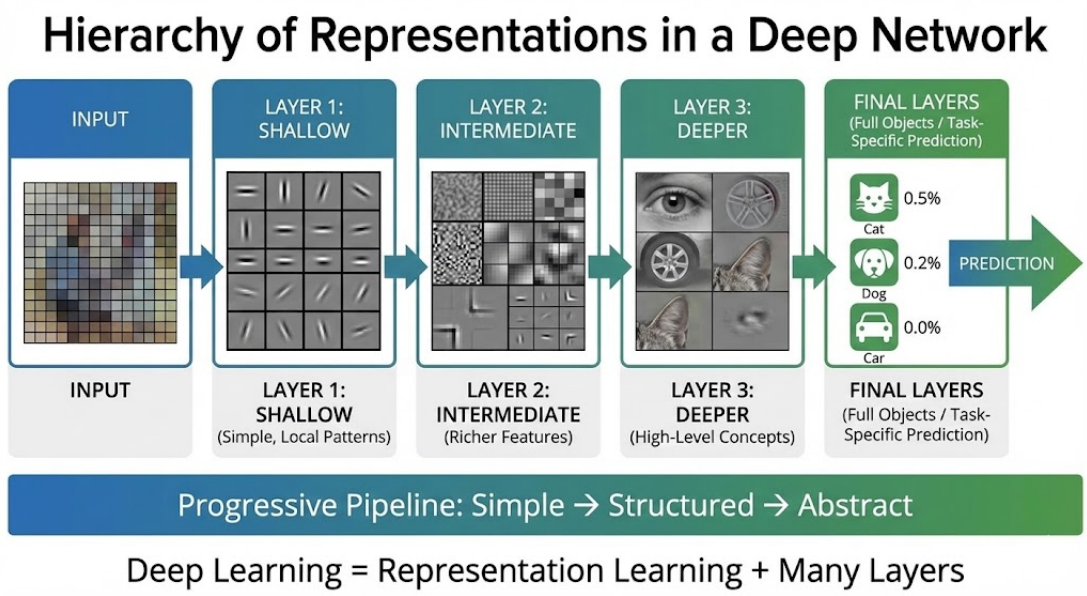

Figure 4.3: **Deep Learning as a Hierarchy of Representations.** Visualizing the "Simple → Abstract" pipeline. While the Input Layer receives raw data (x), the sequence of Hidden Layers functions as a progressive filter. Early layers detect simple patterns (like edges), while deeper layers combine these into high-level concepts, allowing the network to learn a complex representation of the data before making a final prediction (y).

In image (4.3) classification, this hierarchy often looks like:

- Layer 1: horizontal/vertical edges
- Layer 2: patterns, textures
- Layer 3: motifs and object parts
- Layer 4+: full objects (cat, dog, car, etc.)

Deep learning therefore means:

$$\textbf{representation learning} \; + \; \textbf{many layers}.$$

The network is not just fitting a function; it is *learning a sequence of increasingly meaningful representations of the data*.

4.7 Initialization and Training Dynamics

Before training begins, all weights in a neural network must be initialized[1]. Proper initialization is **not a minor technical detail**: it determines whether gradients propagate correctly, whether activations explode or vanish, and whether learning remains stable throughout the network. A poorly initialized network may fail to learn *even if the architecture is correct*.

Why initialization matters

Consider a layer:

$$z^{(l)} = W^{(l)} h^{(l-1)} + b^{(l)}, \qquad h^{(l)} = \sigma\left(z^{(l)}\right).$$

If the weights in $W^{(l)}$ are:

- **too small**: then all $z^{(l)} \approx 0$, and activations saturate or collapse. Gradients become tiny $\rightarrow$ *vanishing gradients*.
- **too large**: then $z^{(l)}$ becomes very large, pushing activations into extreme values. Gradients explode $\rightarrow$ training becomes unstable.

Thus, initialization must keep:

$$\mathrm{Var}(h^{(l)}) \quad \text{and} \quad \mathrm{Var}(z^{(l)})$$

approximately constant across layers.

Key Idea

Proper initialization keeps the signal (forward pass) and gradients (backward pass) within a healthy range throughout the network.

Xavier and He Initialization

Modern initialization schemes choose the variance of the weights so that activations neither shrink nor blow up when passing through many layers.

Xavier (Glorot) Initialization Designed for sigmoid/tanh networks. Weights are drawn such that:

$$W_{ij}^{(l)} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right)$$

or uniformly in a symmetric interval with the same variance.

[1] This discussion follows the presentation in J. D. Prince, *Understanding Deep Learning* (2023), Chapters 11–12.

This keeps the forward and backward variances roughly balanced.

He Initialization Designed for ReLU networks. ReLU discards half of the inputs, so we need larger variance:

$$W_{ij}^{(l)} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right).$$

Used in almost all modern CNNs and MLPs with ReLU.

Note

Choice of activation $\rightarrow$ choice of initialization. Xavier: sigmoid / tanh He: ReLU and variants (LeakyReLU, GELU)

Dynamics During Training

Training is governed by gradient descent, but the *geometry of the loss landscape* strongly affects convergence. Prince identifies three phenomena:

- **Shattered gradients** In deep random networks, gradients may become uncorrelated across layers. Proper initialization mitigates this.
- **Saddle points** Points where the gradient is nearly zero but the model is not optimal. Common in high-dimensional spaces.
- **Flat and sharp minima** Flat minima generalize better; sharp minima tend to overfit. Large-batch training often leads to sharp minima; small batches help find flatter ones.

Key Idea

Training dynamics depend on the interplay between initialization, activation functions, the optimizer, and the loss landscape. Initialization sets the learning trajectory before the first gradient step.

Practical Guidelines

- Use **He initialization** for ReLU networks.
- Use **Xavier initialization** for tanh/sigmoid networks.
- Always check whether gradients vanish or explode in early iterations.
- If training diverges, reduce learning rate or inspect initialization.
- Monitor variance of activations across layers (Prince calls this *activation diagnostics*).

Note

A deep model may fail not because of architecture or data, but because the initial weights sent gradients into a bad regime. Correct initialization is the first step of successful training.

4.8 Overfitting and Regularization in Neural Networks

Neural networks often contain millions of parameters. This makes them extremely flexible, but also highly prone to **overfitting**: the network may memorize the training data instead of learning the underlying patterns. When this happens, training loss becomes very small while test loss increases.

To prevent overfitting, several regularization strategies are commonly used.

Regularization Techniques

- **L2 Weight Decay** Adds a penalty term $\lambda \|W\|_2^2$ to the loss. This discourages large weights and encourages smoother, simpler models. It is mathematically equivalent to a Gaussian prior over the parameters.
- **Dropout** During training, each neuron is randomly "dropped" (set to zero) with a fixed probability p. This prevents neurons from relying too strongly on one another, forcing the network to learn redundant and more robust features.
- **Early Stopping** Training is halted when the validation loss stops improving. This prevents the model from continuing to fit noise in the training set after it has already learned the useful structure.
- **Batch Normalization** Normalizes activations within each mini-batch. This reduces internal covariate shift, stabilizes gradients, and often has a regularizing effect by adding small noise due to batch statistics.

Key Idea

Dropout works by randomly disabling neurons during training. A neuron is kept with probability $(1-p)$ and removed with probability p. Because each training pass uses a different subset of neurons, the network cannot rely on specific activations and must learn more stable, general features.

4.9 Python Example: Neural Network for Classification (Keras)

Below is a minimal neural network for binary classification (breast cancer dataset).

Code Example

Neural Network (Keras)

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

# Load data
data = load_breast_cancer()
X, y = data.data, data.target

# Scale features
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Build model
model = Sequential([
    Dense(16, activation='relu', input_shape=(X_train.shape[1],)),
    Dense(8, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compile model
model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='binary_crossentropy',
    metrics=['accuracy']
)

# Train
model.fit(X_train, y_train, epochs=20, batch_size=16, verbose=1)

# Evaluate
loss, acc = model.evaluate(X_test, y_test)
print("Accuracy:", acc)
```

The following example illustrates a **Deep Learning** model, specifically a **Convolutional Neural Network (CNN)**, which is widely used in image processing. Unlike a simple MLP, a CNN exploits the spatial structure of images.

- **Conv2D layers** detect patterns such as edges, corners, textures.
- **MaxPooling** reduces spatial resolution while keeping important features.
- **Flatten** converts the feature maps into a vector.
- **Dense layers** perform the final classification into digits (0–9).

Deep networks learn features automatically, layer by layer: from pixels → edges → shapes → digits.

Code Example

Deep Learning Example — Convolutional Neural Network (MNIST)

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.optimizers import Adam

# 1. Load dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# 2. Preprocessing
# CNNs expect 4D inputs: (batch, height, width, channels)
X_train = X_train.reshape(-1, 28, 28, 1) / 255.0
X_test  = X_test.reshape(-1, 28, 28, 1) / 255.0

# Convert labels to one-hot vectors (10 classes: 0-9)
y_train = to_categorical(y_train, 10)
y_test  = to_categorical(y_test, 10)

# 3. Build CNN model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D((2, 2)),

    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),

    Flatten(),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])

# 4. Compile
model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)

# 5. Train
model.fit(X_train, y_train, epochs=5, batch_size=64, verbose=1)

# 6. Evaluate
loss, acc = model.evaluate(X_test, y_test)
print("Test accuracy:", acc)
```

4.10 Summary: Recognizing Neural Networks

Example-Recap

Neural Networks vs. Classical Machine Learning Models

The models presented in the previous chapters (Decision Trees, Random Forests, Gradient Boosting, XGBoost) follow very different principles than neural networks. This comparison clarifies *when* each family of models is appropriate and *what makes them fundamentally different*.

- **Decision Trees**
  - split the feature space using simple rules ($x_j \leq t$),
  - easy to interpret and visualize,
  - operate well on tabular data with mixed data types,
  - but unstable (small changes in data $\rightarrow$ different tree).
- **Random Forest**
  - an ensemble of many decision trees trained on random subsets of data and features,
  - reduces variance and overfitting,
  - robust, reliable baseline for structured/tabular datasets,
  - still limited in modelling very complex high-dimensional data.
- **Gradient Boosting / XGBoost**
  - sequentially build trees that correct previous errors,
  - often state-of-the-art for regression and classification on tabular datasets,
  - excellent performance with good hyperparameter tuning,
  - very strong on small/medium datasets where deep learning may overfit.
- **Neural Networks (MLP)**
  - learn *representations* through multiple layers,
  - handle high-dimensional and dense input spaces (e.g. 1000+ features),
  - approximate complex nonlinear functions (Universal Approximation Theorem),
  - require careful training (initialization, learning rate, normalization).
- **Deep Learning**
  - simply means: *many layers* (deep architectures),
  - hierarchical learned features:

$$\text{raw input} \rightarrow \text{simple features} \rightarrow \text{complex patterns}$$

  - excels in images (CNNs), audio, text (Transformers), and large-scale problems,

  - needs:
    * large training datasets,
    * high computational power (GPUs),
    * regularization (dropout, weight decay),
    * careful tuning of optimization and architecture.

Summary of strengths:

- Decision Trees / Random Forest / XGBoost → best for **structured tabular data**, interpretable rules, smaller datasets.
- Neural Networks → best for **unstructured high-dimensional data** (images, audio, text) and complex nonlinear relationships.
- Deep Learning → automatic feature learning, powerful scaling, but requires data and computation.

Summary of challenges:

- tree-based models: can struggle with extremely high-dimensional continuous inputs,
- neural networks: sensitive to initialization, learning rate, overfitting; harder to interpret, need more data.

5

Practical ML

5.1 The Practical Machine Learning Pipeline

Machine learning is not just about choosing an algorithm. A complete ML project follows a structured workflow:

1. **Define the problem** (regression, classification, forecasting, ranking, etc.).
2. **Collect data** and ensure it represents the real task.
3. **Explore the data** (summary statistics, correlations, distributions).
4. **Preprocess**:
    - handle missing values,
    - encode categorical variables,
    - scale or normalize features when required,
    - remove or cap outliers.
5. **Split** into train / validation / test.
6. **Choose a baseline model** (linear, tree-based, etc.).
7. **Train the model** using the training set.
8. **Tune hyperparameters** using the validation set or cross-validation.
9. **Evaluate** using appropriate metrics.
10. **Deploy** and monitor performance over time.

Key Idea

A practical ML project is mostly data work, not model selection. Choosing the algorithm is often less important than preparing the data well.

5.2 Evaluation Metrics

Different tasks require different metrics. Choosing the wrong metric leads to misleading conclusions.

Regression Metrics

- **MSE** (mean squared error): penalizes large errors strongly.
- **RMSE**: interpretable in the same units as the target.
- **MAE**: more robust to outliers.
- R^2: proportion of variance explained by the model.

Classification Metrics

- **Accuracy**: fraction of correctly classified samples.
- **Precision**: out of positive predictions, how many were correct?
- **Recall**: out of actual positives, how many were detected?
- **F1-score**: harmonic mean of precision and recall.
- **ROC–AUC**: probability that the classifier ranks a random positive higher than a random negative.

> Note
>
> In imbalanced datasets (fraud detection, medical diagnosis), accuracy is misleading. Recall, precision, and F1-score are essential.

5.3 Hyperparameter Tuning and Model Selection

Hyperparameters control how a model learns. Examples:

- learning rate (neural networks, boosting),
- max_depth (trees, forests, boosting),
- regularization strength (ridge, lasso, logistic regression).

Search Strategies

Grid Search Tests all combinations of hyperparameters.

Random Search Samples random configurations; often more efficient.

Bayesian Optimization Builds a probabilistic model of performance and explores promising regions. Useful for expensive models (deep learning, XGBoost).

Key Idea

Hyperparameters are not learned from data. They must be chosen using the validation set or cross-validation.

5.4 Data Preprocessing Cheatsheet

Scaling

Needed for: linear models, logistic regression, SVMs, neural networks. Not needed for: trees, random forests, boosting.

Encoding Categorical Features

- One-hot encoding (most common).
- Ordinal encoding (when categories have natural order).
- Target encoding (careful: risk of leakage).

Handling Missing Data

- mean/median imputation,
- KNN imputation,
- model-based imputation,
- using “missing” as its own category (trees often benefit).

Note

Most ML performance differences come from preprocessing, not from the choice of algorithm.

Annex: Machine Learning Interview Q&A

This annex collects short question-and-answer pairs on key concepts from these notes.

1. Foundations

Q1. What is the difference between supervised and unsupervised learning?

A: Supervised learning uses labelled data (x_i, y_i) to learn a function $f_\theta(x)$ that predicts y from x. Unsupervised learning only uses inputs x_i and tries to discover structure in the data (clusters, low-dimensional representations, density).

Q2. Explain the bias–variance tradeoff.

A: The expected squared prediction error decomposes as

$$\mathbb{E}[(y - \hat{f}(x))^2] = \text{Bias}[\hat{f}(x)]^2 + \text{Var}[\hat{f}(x)] + \sigma^2_{\text{noise}}.$$

- High bias: model too simple, underfits.
- High variance: model too flexible, overfits.

Good models balance both.

Q3. What assumptions does linear regression make?

A:

- The true relationship is approximately linear: $y = X\beta + \varepsilon$.
- Errors ε are independent, zero mean, constant variance, often assumed Gaussian.
- Features are not perfectly collinear.

2. Loss, Risk, and Optimization

Q4. What is a loss function?

A: A loss function $L(y, \hat{y})$ measures how far a prediction $\hat{y}$ is from the true target y. Training means choosing θ to minimise the average loss over the data.

Q5. What is empirical risk?

A: For data $(x_i, y_i)_{i=1}^n$ and model f_θ,

$$R_{\text{emp}}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell\big(y_i, f_\theta(x_i)\big)$$

is the empirical risk, the average loss on the training set. We minimise R_{emp} because the true risk $R(\theta) = \mathbb{E}_{(x,y)\sim P_{\text{data}}}[\ell(y, f_\theta(x))]$ is not directly observable.

Q6. Why is cross-entropy used for classification?

A: For a Bernoulli or categorical model, cross-entropy is exactly the negative log-likelihood of the correct class. Minimising cross-entropy is equivalent to maximising likelihood and encourages the model to place high probability on the true label.

Q7. What is gradient descent?

A: An iterative optimisation method:

$$\theta \leftarrow \theta - \eta \nabla_\theta L(\theta),$$

where η is the learning rate. Variants: stochastic gradient descent (SGD), Adam, RMSProp, momentum.

3. Train / Validation / Test and Evaluation

Q8. Why do we split data into train, validation, and test sets?

A:

- Train set: fit model parameters.
- Validation set: tune hyperparameters, choose the model.
- Test set: estimate final performance on unseen data.

This prevents using the same data both for fitting and for evaluation, which would give an over-optimistic estimate.

Q9. What is k-fold cross-validation?

A: We split the data into k folds. For each fold: train on $k-1$ folds and validate on the remaining fold. Average the validation scores. This gives a more stable estimate, especially when data is limited.

4. Trees, Random Forests, and Boosting

Q10. How does a decision tree choose a split?

A: At each node, it considers candidate splits of the form $x_j \leq t$ and chooses the split that maximises impurity reduction:

- classification: reduce Gini or entropy,
- regression: reduce mean squared error.

Q11. Why are random forests effective?

A: They average many de-correlated trees:

- each tree is trained on a bootstrap sample of the data,
- at each split, only a random subset of features is considered.

This reduces variance and overfitting compared to a single tree.

Q12. What is gradient boosting in simple terms?

A: Gradient boosting builds a sequence of trees

$$F_m(x) = F_{m-1}(x) + \nu h_m(x),$$

where each new tree h_m is trained to fit the residuals (or negative gradient of the loss) of the current model. Each step corrects errors of the previous steps.

Q13. What makes XGBoost special compared to basic gradient boosting?

A:

- Uses second-order Taylor expansion of the loss (gradients and Hessians).
- Includes explicit regularisation on tree complexity (L1 and L2).
- Very efficient implementation: sparse-aware, parallel, out-of-core.
- Handles missing values and column subsampling.

5. Neural Networks and Deep Learning

Q14. What is the role of an activation function?

A: Activation functions $\sigma(z)$ introduce nonlinearity into each neuron. Without them, stacking layers would still yield a linear function $f(x) = Wx + b$, equivalent to linear regression.

Q15. What is backpropagation?

A: Backpropagation is an efficient application of the chain rule to compute gradients of the loss with respect to all parameters in a network. For layer l,

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial h^{(l)}} \cdot \frac{\partial h^{(l)}}{\partial z^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial W^{(l)}}.$$

Q16. Why is weight initialization important in deep networks?

A: If initial weights are too small, activations and gradients vanish. If too large, they explode. Proper schemes (Xavier, He initialization) keep the variance of activations and gradients roughly stable across layers, allowing effective training.

Q17. Name common regularisation techniques for neural networks.

A:

- L2 weight decay,
- dropout,
- early stopping using validation loss,
- batch normalization (indirect regularisation).

6. Practical Interview Scenarios

Q18. Your model has low training error but high test error. What does this indicate?

A: This suggests overfitting. Possible actions:

- add regularisation (L2, dropout),
- gather more data or augment existing data,
- simplify the model (fewer parameters, smaller depth),
- improve train/validation split and cross-validation.

Q19. Your model has high training and high test error. What does this indicate?

A: This suggests underfitting. Possible actions:

- use a more expressive model,
- add features or use feature engineering,
- reduce regularisation,
- check data preprocessing and labels for errors.

Q20. You have a small tabular dataset. Would you use deep learning or XGBoost?

A: Usually XGBoost or Random Forest is preferred:

- tree-based models handle small datasets well,
- they need less tuning and less data,
- deep networks tend to overfit and need more data and computation.

Deep learning becomes attractive for large, high-dimensional, or unstructured data.

7. Advanced / Difficult Interview Questions

Q21. What is the difference between MLE and MAP in simple terms?

A:

- MLE chooses the parameter θ that makes the observed data most likely.
- MAP chooses the parameter θ that is most likely *after combining* the data (likelihood) and prior knowledge (prior).

MAP = MLE + regularisation.

Q22. What is the KL divergence and why is it important?

A: The Kullback–Leibler divergence measures how different two probability distributions are:

$$D_{\mathrm{KL}}(P\|Q) = \mathbb{E}_{x\sim P}\left[\log \frac{P(x)}{Q(x)}\right].$$

It appears in variational inference, VAEs, reinforcement learning, and regularisation.

Q23. What is the vanishing gradient problem?

A: During backpropagation, gradients are products of many derivatives. If these derivatives are small (for example using sigmoid or tanh), their products shrink exponentially, so early layers receive almost zero gradient and stop learning.

Q24. Why does batch normalisation help training?

A: It rescales activations inside the network to have stable mean and variance. This:

- reduces internal covariate shift,
- stabilises the gradient flow,
- allows higher learning rates,
- provides regularisation.

Q25. What is the difference between bagging and boosting?

A:

- Bagging (Random Forest): trains many models independently on bootstrapped samples and averages them to reduce variance.
- Boosting (Gradient Boosting/XGBoost): builds models *sequentially*, each correcting errors of the previous model.

Q26. When is deep learning a bad choice?

A:

- small datasets,
- tabular data with strong feature semantics,
- limited compute or tight latency constraints,
- when interpretability is essential.

Q27. What is regularisation from an optimisation viewpoint?

A: Regularisation adds a penalty $\lambda\Omega(\theta)$ to the loss function. This constrains the parameter search space and reduces variance by discouraging complex solutions.

Q28. Why is XGBoost faster than traditional gradient boosting?

A:

- parallelised tree construction,
- efficient cache-aware implementation,
- handling of sparsity and missing values,
- second-order gradients (both gradient and Hessian),
- regularisation built into tree growth.

Q29. How do neural networks approximate arbitrary functions?

A: With enough neurons and a nonlinear activation, a single hidden layer can approximate any continuous function (Universal Approximation Theorem). Depth allows more efficient and structured representations.

Q30. What is early stopping and why does it work?

A: It stops training when validation loss stops improving. This prevents the model from overfitting the training data and acts as an implicit regulariser.

Q31. Mathematically, why does minimizing Mean Squared Error (MSE) correspond to maximizing the likelihood of Gaussian data?

A: Assume the target y is generated by a deterministic function of x plus Gaussian noise:

$$y_i = f(x_i;\theta) + \varepsilon_i, \quad \varepsilon_i \sim \mathcal{N}(0,\sigma^2).$$

The probability density for a single observation is:

$$p(y_i|x_i;\theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - f(x_i;\theta))^2}{2\sigma^2}\right).$$

The log-likelihood for n independent samples is:

$$\log \mathscr{L}(\theta) = \sum_{i=1}^{n} \left[-\frac{1}{2}\log\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}(y_i - f(x_i;\theta))^2\right].$$

To maximize this, we must minimize the term with the negative sign. Since σ is constant, this is equivalent to minimizing:

$$\sum_{i=1}^{n}(y_i - f(x_i;\theta))^2,$$

which is the Sum of Squared Errors.

Q32. Why does L1 regularization (Lasso) yield sparse solutions (zeros), while L2 (Ridge) does not?

A:

- **Geometric view:** L1 regularization constrains the parameters to a diamond shape ($|\theta_1| + |\theta_2| \leq C$). The error contours (ellipses) of the loss function are likely to touch the "comers" of this diamond, where coefficients are exactly zero. L2 constrains to a circle ($\theta_1^2 + \theta_2^2 \leq C$), where the touch point is rarely on an axis.
- **Probabilistic view:** L1 corresponds to a **Laplace prior** (sharp peak at zero). L2 corresponds to a **Gaussian prior** (smooth bell curve). The sharp peak pulls the MAP estimate strongly to exactly zero.

Q33. In an ensemble, why does Bagging reduce variance while Boosting reduces bias?

A:

- **Bagging (e.g., Random Forest):** Averages T independent low-bias, high-variance models. If errors are uncorrelated with variance σ^2, averaging reduces variance to σ^2/T. It cannot reduce bias because the average of biased models is still biased.
- **Boosting:** Starts with a high-bias model (weak learner). Each subsequent model attempts to correct the residual error of the previous ones. This iterative correction progressively reduces the systematic error (bias), though it can increase variance if it overfits the noise.

Q34. How does Batch Normalization behave differently during training vs. prediction (inference)?

A:

- **During Training:** It normalizes the layer inputs using the mean μ_B and variance σ_B^2 of the *current mini-batch*. This allows the network to learn stable distributions.
- **During Inference:** We process single samples, so we cannot compute a batch mean. Instead, we use the *running average* of the means and variances collected during training.

A common interview failure is forgetting that inference uses fixed population statistics, not batch statistics.

Q35. Why are saddle points a bigger problem than local minima in high-dimensional optimization?

A: In high dimensions, it is statistically very unlikely for a critical point (zero gradient) to be a local minimum. For a point to be a minimum, the curvature (eigenvalues of the Hessian) must be positive in *all* dimensions. It is much more likely that the curvature is positive in some directions and negative in others (a saddle point). Gradient descent can get "stuck" slowing down near saddle points, which is why momentum-based optimizers are useful.

Q36. Why do Sigmoid and Tanh activations cause the "Vanishing Gradient" problem?

A: The derivative of the Sigmoid function is $\sigma'(z) = \sigma(z)(1-\sigma(z))$. The maximum value of this derivative is 0.25 (at $z=0$). In a deep network, backpropagation multiplies these derivatives via the chain rule.

$$\text{Gradient} \propto 0.25 \times 0.25 \times 0.25 \ldots$$

As depth increases, the gradient exponentially decays to zero, stopping the weights in early layers from updating. ReLU solves this because its derivative is exactly 1 for positive inputs.

Q37. How would you handle an extremely imbalanced dataset (e.g., 1% fraud)?

A: Accuracy is a useless metric here. Strategies include:

- **Metrics:** Use Precision, Recall, F1-Score, or AUC-PR (Area Under Precision-Recall Curve).
- **Resampling:** Undersample the majority class or oversample the minority class (SMOTE).
- **Cost-sensitive learning:** Modify the loss function to penalize errors on the minority class more heavily (e.g., `class_weight='balanced'` in scikit-learn).
- **Focal Loss:** A specific loss function (popular in object detection) that down-weights easy examples and focuses training on hard (minority) examples.

Q38. What is the complexity of the Self-Attention mechanism in Transformers?

A: For a sequence of length N and dimension d: Calculating the attention scores involves multiplying matrices $(N \times d)$ and $(d \times N)$, resulting in an $(N \times N)$ matrix. Therefore, the time and memory complexity is $**O(N^2)**$ (quadratic). This is why standard Transformers struggle with very long documents compared to RNNs (linear $O(N)$) or CNNs.

8. High-Technical Questions (for Deep Understanding)

Q39. Show that least–squares linear regression is the MLE under Gaussian noise.

A: Assume the model

$$y_i = f_\theta(x_i) + \varepsilon_i, \qquad \varepsilon_i \sim \mathcal{N}(0, \sigma^2), \text{ independent.}$$

For linear regression, $f_\theta(x_i) = x_i^\top w$. The likelihood of the data is

$$p(y \mid X, w) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - x_i^\top w)^2}{2\sigma^2} \right).$$

The log-likelihood is

$$\log p(y \mid X, w) = -\frac{n}{2} \log\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - x_i^\top w)^2.$$

Maximising this over w is equivalent to minimising

$$\sum_{i=1}^{n}(y_i - x_i^\top w)^2,$$

which is exactly the least–squares objective. Therefore OLS is the maximum likelihood estimator under Gaussian noise.

Q40. Why is cross-entropy equal to negative log-likelihood in logistic regression?

A: For a binary label $y_i \in \{0,1\}$ and predicted probability $\hat{p}_i = \sigma(w^\top x_i)$, the Bernoulli likelihood is

$$p(y_i \mid x_i, w) = \hat{p}_i^{y_i}(1-\hat{p}_i)^{1-y_i}.$$

The log-likelihood of the dataset is

$$\log \mathscr{L}(w) = \sum_{i=1}^{n} \left[y_i \log \hat{p}_i + (1-y_i)\log(1-\hat{p}_i) \right].$$

The cross-entropy loss is defined as

$$\mathscr{L}_{\mathrm{CE}}(w) = -\sum_{i=1}^{n} \left[y_i \log \hat{p}_i + (1-y_i)\log(1-\hat{p}_i) \right] = -\log \mathscr{L}(w).$$

Thus minimising cross-entropy is exactly the same as maximising the log-likelihood.

Q33. Show that logistic regression has a linear decision boundary.

A: Logistic regression predicts

$$P(y=1 \mid x) = \sigma(w^\top x + b).$$

A common decision rule is to predict class 1 if this probability is at least 0.5:

$$P(y=1 \mid x) \geq 0.5 \quad \Longleftrightarrow \quad \sigma(w^\top x + b) \geq 0.5.$$

Since $\sigma(z)$ is strictly increasing and $\sigma(0) = 0.5$, this is equivalent to

$$w^\top x + b \geq 0.$$

The set of points satisfying $w^\top x + b = 0$ is a hyperplane in feature space, so the decision boundary is linear.

Q41. In gradient boosting, how does "fit the negative gradient" generalise the idea of fitting residuals?

A: Let the model at step m be $F_{m-1}(x)$. We want to minimise an empirical risk

$$R(F) = \frac{1}{n}\sum_{i=1}^{n} L\big(y_i, F(x_i)\big).$$

The functional gradient with respect to the current predictions at the points x_i is

$$g_i^{(m)} = \left.\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right|_{F=F_{m-1}}.$$

Gradient descent in function space would update

$$F_m(x_i) = F_{m-1}(x_i) - \nu\, g_i^{(m)}.$$

Gradient boosting approximates the negative gradient $-g_i^{(m)}$ by a tree $h_m(x)$ fitted to the targets $-g_i^{(m)}$. For squared error,

$$L(y, F(x)) = (y - F(x))^2 \quad \Rightarrow \quad g_i^{(m)} = -2\,(y_i - F_{m-1}(x_i)),$$

so fitting $-g_i^{(m)}$ is equivalent to fitting the residuals $y_i - F_{m-1}(x_i)$. Thus "fit residuals" is a special case of "fit negative gradients".

Q42. Why does a network without activation functions reduce to linear regression, even with many layers?

A: Consider a network with L layers and no nonlinearities:

$$\begin{aligned} h^{(1)} &= W^{(1)}x + b^{(1)}, \\ h^{(2)} &= W^{(2)}h^{(1)} + b^{(2)}, \\ &\vdots \\ \hat{y} &= W^{(L)}h^{(L-1)} + b^{(L)}. \end{aligned}$$

By substituting step by step, we obtain

$$\hat{y} = W^{(L)}W^{(L-1)}\cdots W^{(1)}x + b^* = W^*x + b^*,$$

for some effective weight matrix W^* and bias b^*. This is again just a linear map from x to $\hat{y}$, the same form as linear regression. Nonlinear activation functions are therefore essential to obtain nonlinear decision boundaries and expressive power.