
MACHINE LEARNING
with Real-World Applications using
PYTHON
Dr. Muhammad Siddique

Machine Learning with Real-World Applications using Python

January 30, 2026

Copyright

Published by Muhammad Siddique
Email: msiddique@nfciet.edu.pk

Contents

List of Figures

List of Tables

Part I

Machine Learning

Chapter 1

Introduction to Machine Learning

The Central Question of Learning

How can a machine improve its performance by learning from experience rather than explicit programming?

Machine Learning (ML) is a core discipline within Artificial Intelligence that focuses on the design of algorithms capable of **learning from data**, identifying hidden patterns, and improving decision-making performance over time. Unlike traditional rule-based software=where every behavior must be explicitly programmed=machine learning systems automatically construct internal models by analyzing historical data and extracting statistical regularities.

At its essence, machine learning transforms **data into knowledge**. An ML system does not merely store examples; it generalizes from them, enabling intelligent behavior in previously unseen situations. This method shift has revolutionized computing by allowing machines to operate effectively in environments that are too complex, uncertain, or dynamic to be handled by hand-crafted rules.

Why Machine Learning Matters Machine learning underpins many of the most impactful technologies of the modern era, including:

- **Speech and Language Processing:** Voice assistants, machine translation, sentiment analysis.
- **Computer Vision:** Facial recognition, medical imaging, autonomous driving.

- **Decision Support Systems:** Credit scoring, fraud detection, clinical diagnosis.
- **Recommendation Engines:** Personalized content on streaming platforms and e-commerce.

Key Insight

Traditional programming tells the computer *how to act*. Machine learning allows the computer to *discover how to act*.

1.1 Introduction

Machine learning algorithms are broadly classified into three fundamental **learning methods** based on the nature of the feedback available to the learner:

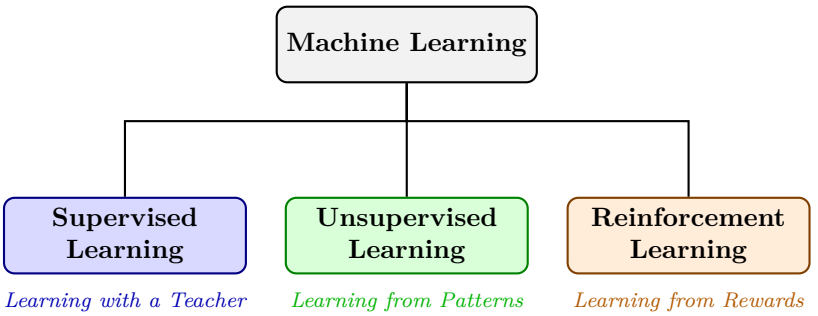


Figure 1.1: Taxonomy of Machine Learning methods.

Each method addresses a different class of real-world problems and imposes distinct assumptions about the availability of ground truth.

Taxonomy of Learning Methods

Method	Type of Feedback	Typical Applications
Supervised	Labeled data (Ground truth)	Prediction, classification, regression
Unsupervised	No labels (Data structure only)	Clustering, dimensionality reduction, pattern discovery
Reinforcement	Reward/Penalty signals	Control systems, robotics, strategic games

1.2 Supervised Learning

Learning from Labeled Examples

Supervised learning enables machines to learn a mapping between inputs and known outputs using labeled data.

Supervised Learning is the most fundamental and widely adopted learning method in machine learning. In this approach, the learning algorithm is provided with a dataset consisting of **input–output pairs**, where each input example is explicitly associated with a correct target label. The presence of labeled data allows the model to directly quantify its prediction error and iteratively improve its performance.

From a conceptual standpoint, supervised learning closely resembles human learning in a classroom setting, where a student learns by comparing their answers with the correct solutions provided by a teacher.

Supervised learning is extensively used in applications where historical labeled data is available and accurate prediction is required, such as medical diagnosis, credit risk assessment, speech recognition, and image classification.

Mathematical Formulation

Formally, a supervised learning problem can be defined as follows:

- Let the input space be $X \subset \mathbb{R}^n$, representing n features.
- Let the output space be Y, which may be discrete (for classification) or continuous (for regression).
- Given a training dataset:

$$\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\}$$

the goal is to learn a function $f : X \to Y$ that minimizes a chosen loss function $L(y, \hat{y})$, where $\hat{y} = f(x)$ is the predicted output.

Common loss functions include:

1. Mean Squared Error (MSE) for regression:

$$L(y, \hat{y}) = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

2. Cross-Entropy Loss for classification:

$$L(y, \hat{y}) = -\sum_{i=1}^{m} y_i \log(\hat{y}_i)$$

Interpretation

The loss function acts as a quantitative measure of how "wrong" the model's predictions are.

Types of Supervised Learning Problems

Supervised learning problems are broadly classified into two main categories:

1.2.1 Regression Problems

Regression aims to predict a **continuous-valued output**. Examples include house price prediction, temperature forecasting, and stock price estimation.

Numerical Case Study: Linear Regression (Step-by-Step)

House Price Prediction Problem

A real-world regression example to understand supervised learning numerically.

Step 1: Dataset Suppose we have the following labeled dataset:

$$X = [1000,\ 1500,\ 2000] \quad \text{(Area in sq.ft)}$$

$$Y = [150,\ 200,\ 250] \quad \text{(Price in thousand USD)}$$

Step 2: Model Assumption We assume a linear relationship between area and price:

$$\hat{Y} = wX + b$$

where:

- w is the weight (slope),
- b is the bias (intercept).

Step 3: Learning Parameters After training (e.g., using gradient descent), suppose the learned parameters are:

$$w = 0.1, \quad b = 50$$

Thus, the final model becomes:

$$\hat{Y} = 0.1X + 50$$

Step 4: Prediction on New Data For a new house with area $X = 1800$ sq.ft:

$$\hat{Y} = 0.1 \times 1800 + 50 = 230$$

Interpretation

The model successfully generalizes learned knowledge to an unseen example, which is the primary goal of supervised learning.

1.2.2 Classification Problems

Classification tasks aim to predict a **discrete or categorical label** based on input features. Unlike regression, where outputs are continuous, classification outputs belong to a finite set of classes. Common real-world applications include spam email detection, medical diagnosis, credit risk assessment, sentiment analysis, and image recognition.

Typical classification algorithms include Decision Trees, k-Nearest Neighbors (KNN), Support Vector Machines (SVM), Logistic Regression, and Neural Networks. These models learn decision boundaries that separate different classes in the feature space.

Numerical Case Study: Binary Classification Using KNN

Disease Diagnosis Problem

A Example to understand supervised classification using a simple KNN classifier.

Step 1: Dataset Consider a small labeled dataset where the goal is to predict whether a patient has a disease based on their blood pressure readings:

$$X = [120,\ 140,\ 130,\ 160] \quad \text{(Blood Pressure in mmHg)}$$

$$Y = [0,\ 1,\ 0,\ 1] \quad (0 = \text{Healthy},\ 1 = \text{Diseased})$$

Here, each input value corresponds to a known class label, making this a supervised learning problem.

Step 2: Model Selection We select the **k-Nearest Neighbors (KNN)** algorithm for classification. KNN is a non-parametric, instance-based learning method that classifies a new data point based on the majority label of its k closest neighbors in the training dataset.

For simplicity, assume:

$$k = 3$$

Step 3: Distance Computation Suppose a new patient arrives with a blood pressure reading of:

$$X_{\text{new}} = 135$$

We compute the absolute distance between this value and each training sample:

$$|135 - 120| = 15$$

$$|135 - 130| = 5$$

$$|135 - 140| = 5$$

$$|135 - 160| = 25$$

The three nearest neighbors correspond to blood pressure values:

$$130,\ 140,\ 120$$

Step 4: Majority Voting The class labels of the three nearest neighbors are:

$$[0,\ 1,\ 0]$$

Since the majority class is **0 (Healthy)**, the KNN classifier predicts:

$$\hat{Y} = 0$$

Prediction Result

The new patient is classified as Healthy.

By: Engr. Dr. Muhammad Siddique, Postdoc AI (YALE University, USA).
HOD Artificial Intelligence, NFC IET Multan, msiddique@nfciet.edu.pk

Discussion This example illustrates how classification models assign discrete labels rather than continuous values. Even with a simple dataset, the classifier can effectively learn patterns and make decisions based on similarity measures. In real-world applications, classification models often use multiple features (e.g., blood pressure, age, cholesterol level) and more advanced distance metrics to improve accuracy.

Key Insight

Classification focuses on decision boundaries and class separation, making it a fundamental task in supervised learning for decision-making systems.

Regression vs Classification

Aspect	Regression	Classification
Output type	Continuous	Discrete
Example	House price	Spam / Not spam
Common loss	MSE	Cross-entropy

Bias–Variance Intuition

- **High Bias:** Model is too simple, leading to underfitting.
- **High Variance:** Model is too complex, leading to overfitting.

A well-trained supervised learning model achieves a balance between bias and variance to ensure good generalization.

Key Takeaway: Supervised learning is effective because it combines mathematical optimization, statistical inference, and data-driven learning to produce accurate and generalizable models.

1.2.3 Model Evaluation in Supervised Learning

How Do We Measure Model Performance?

A supervised learning model must be evaluated using objective performance metrics.

Once a supervised learning model has been trained, its effectiveness must be evaluated using unseen data. Model evaluation ensures that the learned

function generalizes well beyond the training examples and does not merely memorize the dataset.

Train–Test Split Typically, the available dataset is divided into:

- **Training Set:** Used to learn model parameters.
- **Testing Set:** Used to evaluate model performance.

This separation allows an unbiased estimate of real-world performance.

Evaluation Metrics for Regression

1. Mean Squared Error (MSE)

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

MSE penalizes larger errors more heavily and is widely used for continuous prediction tasks.

2. Root Mean Squared Error (RMSE)

$$\text{RMSE} = \sqrt{\text{MSE}}$$

RMSE has the same unit as the output variable, making it easier to interpret.

Interpretation

Lower MSE or RMSE values indicate better regression performance.

1.2.4 Evaluation Metrics for Classification

Classification performance is commonly assessed using the **confusion matrix**, which compares predicted labels with true labels.

Key Metrics

- **Accuracy:**
$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$
- **Precision:**
$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Recall:**

$$\text{Recall} = \frac{TP}{TP + FN}$$

- **F1-Score:**

$$F1 = \frac{2 \cdot (\text{Precision} \cdot \text{Recall})}{\text{Precision} + \text{Recall}}$$

Key Insight

Accuracy alone may be misleading, especially in imbalanced datasets. Precision and recall provide deeper insight.

Example: Binary Classification

Spam Email Classification

An example to understand classification metrics numerically.

Suppose a spam classifier produces the following results on a test set:

$$TP = 40,\ TN = 50,\ FP = 10,\ FN = 5$$

Metric Computation

$$\text{Accuracy} = \frac{40 + 50}{105} = 0.857$$

$$\text{Precision} = \frac{40}{40 + 10} = 0.80$$

$$\text{Recall} = \frac{40}{40 + 5} = 0.89$$

Interpretation

The classifier is highly effective at detecting spam emails while maintaining reasonable precision.

Overfitting and Underfitting

Generalization vs Memorization

A well-trained supervised model must generalize, not memorize.

Underfitting Occurs when the model is too simple to capture underlying patterns.

- High training error
- High testing error

Overfitting Occurs when the model learns noise and specific details of the training data.

- Very low training error
- High testing error

Goal

The objective of supervised learning is to achieve low error on both training and unseen data.

Regularization in Supervised Learning

Regularization techniques are used to prevent overfitting by penalizing overly complex models.

1. L2 Regularization (Ridge)

$$L = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \lambda \sum w_i^2$$

2. L1 Regularization (Lasso)

$$L = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \lambda \sum |w_i|$$

Effect of Regularization

Regularization improves generalization by discouraging overly complex models.

1.3 Unsupervised Learning

Learning from Unlabeled Data

Unsupervised learning enables machines to discover hidden patterns and structures in data without explicit output labels.

Unsupervised Learning is a fundamental paradigm in machine learning where the learning algorithm is provided with **input data only**, without any corresponding target labels. Unlike supervised learning, there is no direct notion of "correct answers." Instead, the model attempts to infer the underlying structure, distribution, or relationships present in the data.

From a conceptual perspective, unsupervised learning resembles exploratory human learning, where patterns are discovered through observation and experience rather than explicit instruction. The learner organizes information, identifies similarities, and groups related concepts autonomously.

Unsupervised learning is widely used in applications such as customer segmentation, anomaly detection, topic modeling, market basket analysis, dimensionality reduction, and exploratory data analysis, particularly when labeled data is unavailable or expensive to obtain.

Mathematical Formulation

Formally, an unsupervised learning problem can be defined as follows:

- Let the input space be $X \subset \mathbb{R}^n$, representing n features.
- Given an unlabeled dataset:

$$\mathcal{D} = \{x_1, x_2, \ldots, x_m\}$$

- The objective is to learn a structure, representation, or function:

$$f : X \to Z$$

  where Z may represent clusters, latent variables, or lower-dimensional embeddings.

Instead of minimizing prediction error, unsupervised learning optimizes objectives such as similarity, density estimation, variance maximization, or information preservation.

Interpretation

The model learns *how data is organized* rather than predicting predefined outputs.

1.3.1 Types of Unsupervised Learning Problems

Unsupervised learning problems are broadly classified into two major categories:

1. Clustering Problems

Clustering aims to group data points into **homogeneous clusters** such that points within the same cluster are more similar to each other than to points in other clusters. The number of clusters may be predefined or learned automatically.

Common applications include customer segmentation, image grouping, document clustering, and social network analysis.

Typical clustering algorithms include K-Means, Hierarchical Clustering, DBSCAN, and Gaussian Mixture Models (GMM).

Numerical Case Study: K-Means Clustering (Step-by-Step)

Customer Segmentation Problem

A numerical clustering example to understand unsupervised learning.

Step 1: Dataset Suppose we have a simple dataset representing customer annual spending (in thousand USD):

$$X = [2,\ 4,\ 5,\ 10,\ 12,\ 14]$$

Step 2: Choose Number of Clusters Assume:

$$k = 2$$

Step 3: Initialize Centroids Randomly initialize centroids:

$$\mu_1 = 3, \quad \mu_2 = 11$$

Step 4: Assign Data Points Compute distances and assign points to the nearest centroid:

$$\text{Cluster 1: } [2,\ 4,\ 5]$$

$$\text{Cluster 2: } [10,\ 12,\ 14]$$

Step 5: Update Centroids Compute new centroids:

$$\mu_1 = \frac{2+4+5}{3} = 3.67$$

$$\mu_2 = \frac{10+12+14}{3} = 12$$

Interpretation

K-Means identifies natural groupings in the data without any labeled information.

2. Dimensionality Reduction Problems

Dimensionality reduction seeks to represent high-dimensional data using fewer variables while preserving important information. This improves visualization, reduces noise, and enhances computational efficiency.

Common techniques include Principal Component Analysis (PCA), Autoencoders, and t-SNE.

Case Study: Principal Component Analysis (PCA)

Feature Compression Example

Understanding dimensionality reduction numerically.

Geometric Interpretation of PCA (TikZ Illustration)

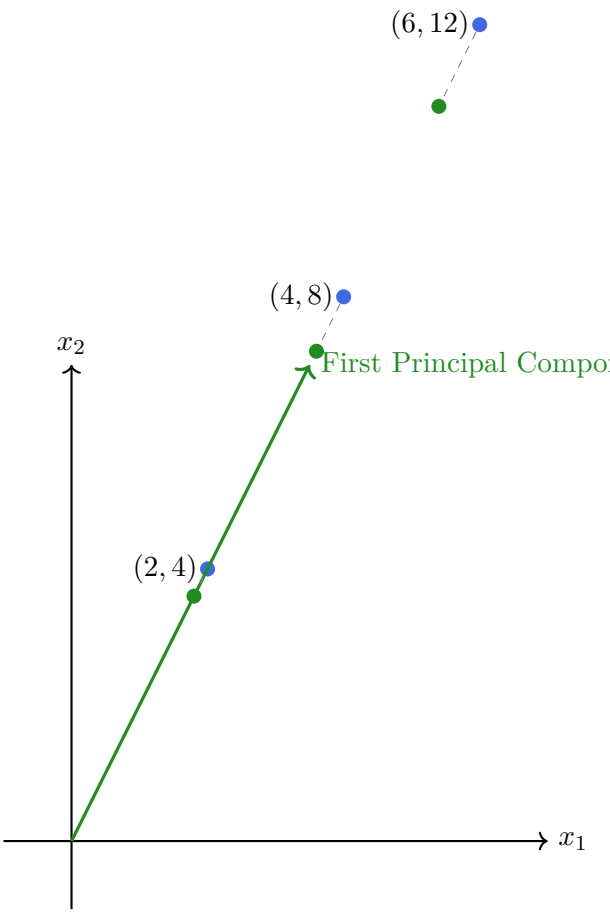


Figure 1.2: Projection of data points onto the first principal component in PCA

Step 1: Dataset Consider a simple dataset consisting of three observations with two correlated features. The second feature is an exact linear multiple of the first, indicating strong redundancy.

$$X = \begin{bmatrix} 2 & 4 \\ 4 & 8 \\ 6 & 12 \end{bmatrix}$$

Here, each row represents a data point, and each column corresponds to a feature. Clearly, the second feature is twice the first, suggesting that the intrinsic dimensionality of the data is one.

Step 2: Mean Centering PCA requires the data to be mean-centered so that variance is measured around the origin.

The mean of each feature is:

$$\mu_1 = \frac{2+4+6}{3} = 4, \qquad \mu_2 = \frac{4+8+12}{3} = 8$$

Subtracting the mean from each observation gives the centered data matrix $\tilde{X}$:

$$\tilde{X} = \begin{bmatrix} -2 & -4 \\ 0 & 0 \\ 2 & 4 \end{bmatrix}$$

Mean centering ensures that PCA captures variance rather than absolute magnitude.

Step 3: Covariance Matrix The covariance matrix summarizes how features vary together. It is computed as:

$$\Sigma = \frac{1}{n}\tilde{X}^T\tilde{X}$$

where $n = 3$ is the number of data points.

First compute $\tilde{X}^T\tilde{X}$:

$$\tilde{X}^T\tilde{X} = \begin{bmatrix} -2 & 0 & 2 \\ -4 & 0 & 4 \end{bmatrix} \begin{bmatrix} -2 & -4 \\ 0 & 0 \\ 2 & 4 \end{bmatrix} = \begin{bmatrix} 8 & 16 \\ 16 & 32 \end{bmatrix}$$

Now divide by n:

$$\Sigma = \begin{bmatrix} \frac{8}{3} & \frac{16}{3} \\ \frac{16}{3} & \frac{32}{3} \end{bmatrix}$$

The large off-diagonal values indicate strong correlation between the two features.

Step 4: Eigen Decomposition PCA identifies principal components by computing the eigenvalues and eigenvectors of the covariance matrix.

Solving

$$\det(\Sigma - \lambda I) = 0$$

yields the eigenvalues:

$$\lambda_1 = \frac{40}{3}, \qquad \lambda_2 = 0$$

The corresponding eigenvectors are:

$$v_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \qquad v_2 = \begin{bmatrix} -2 \\ 1 \end{bmatrix}$$

The eigenvector v_1 associated with the largest eigenvalue λ_1 defines the first principal component, which captures all the variance in the data. The second eigenvalue being zero confirms that the data lies entirely along a single direction.

Step 5: Projection onto the Principal Component To reduce dimensionality, the data is projected onto the first principal component direction:

$$Z = \tilde{X} v_1$$

This projection produces a one-dimensional representation that retains maximum variance while discarding redundant information.

Step 6: Explained Variance The explained variance ratio of the first principal component is:

$$\text{Explained Variance Ratio} = \frac{\lambda_1}{\lambda_1 + \lambda_2} = \frac{\frac{40}{3}}{\frac{40}{3} + 0} = 1$$

This means that **100% of the variance** is preserved using a single principal component.

Key Insight

PCA captures the direction of maximum variance without using any labels, enabling effective dimensionality reduction by removing feature redundancy while preserving essential data structure.

1.3.2 Evaluation in Unsupervised Learning

Unlike supervised learning, evaluation in unsupervised learning is inherently challenging due to the absence of ground truth labels. Instead of direct accuracy measures, performance assessment relies on indirect indicators of structural quality and data organization.

Common Evaluation Techniques Several widely adopted techniques are used to assess unsupervised models, particularly clustering algorithms.

- **Silhouette Score**
- **Elbow Method**

- **Intra-cluster vs Inter-cluster distance**

Challenge

There is no single "correct" answer in unsupervised learning.

Anomaly Detection Perspective

One important application of unsupervised learning is anomaly detection, where the objective is to identify rare or abnormal patterns without relying on labeled examples. In such scenarios, the notion of correctness is defined relative to deviations from learned normal behavior.

- Network intrusion detection
- Fraud detection
- Sensor fault detection

Key Insight

Anomalies are detected as deviations from learned normal patterns.

Bias–Variance Intuition in Unsupervised Learning

Although bias–variance trade-offs are most commonly discussed in supervised learning, similar intuitions apply to unsupervised models when balancing simplicity and sensitivity.

- **High Bias:** Oversimplified structure, missing important patterns.
- **High Variance:** Over-sensitive to noise and minor fluctuations.

A well-designed unsupervised model captures meaningful structure while remaining robust to noise, ensuring that learned representations generalize beyond the training data.

Model Evaluation in Unsupervised Learning

How Do We Evaluate Without Labels?

Evaluating unsupervised learning models requires indirect and structure-based performance measures.

Unlike supervised learning, unsupervised learning does not rely on ground truth labels for evaluation. As a result, model assessment focuses on the

quality of learned structure, **internal consistency**, and **interpretability** of the results rather than direct prediction accuracy.

Evaluation in unsupervised learning often combines mathematical metrics, domain knowledge, and visual inspection to determine whether the discovered patterns are meaningful and useful. The absence of labeled outputs means that there is no single definitive measure of correctness. Consequently, multiple complementary metrics are usually employed.

Internal Evaluation Metrics

Internal metrics assess the quality of the learned structure using only the training data itself, without any external references or annotations.

1. Silhouette Coefficient The silhouette coefficient measures how well a data point fits within its assigned cluster compared to other clusters.

For a data point i:

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))}$$

where:

- $a(i)$ is the average distance between i and points in the same cluster,
- $b(i)$ is the minimum average distance between i and points in other clusters.
- $s(i) \approx 1$: Well-clustered
- $s(i) \approx 0$: Overlapping clusters
- $s(i) < 0$: Misclassified point

Interpretation

Higher silhouette scores indicate better-defined and more separated clusters.

Cluster Compactness and Separation

Beyond silhouette analysis, clustering quality is also evaluated through compactness and separation measures.

1. Intra-Cluster Distance Measures how close data points are within the same cluster:

$$D_{\text{intra}} = \frac{1}{k} \sum_{j=1}^{k} \frac{1}{|C_j|} \sum_{x_i \in C_j} \|x_i - \mu_j\|$$

Lower intra-cluster distance indicates tighter clusters.

2. Inter-Cluster Distance Measures separation between different clusters:

$$D_{\text{inter}} = \min_{i \neq j} \|\mu_i - \mu_j\|$$

Higher inter-cluster distance implies better cluster separation.

Key Insight

A good clustering minimizes intra-cluster distance while maximizing inter-cluster separation.

1.3.3 Model Selection Using the Elbow Method

Choosing the Number of Clusters

Determining model complexity in unsupervised learning.

The **Elbow Method** is commonly used to select the optimal number of clusters k in clustering algorithms such as K-Means, providing a heuristic balance between model complexity and representational fidelity.

Within-Cluster Sum of Squares (WCSS)

$$\text{WCSS}(k) = \sum_{j=1}^{k} \sum_{x_i \in C_j} \|x_i - \mu_j\|^2$$

As k increases, WCSS decreases. The optimal k is chosen at the point where the decrease in WCSS begins to slow significantly (the "elbow point").

Interpretation

The elbow point balances model simplicity and clustering quality.

1.3.4 Evaluation of Dimensionality Reduction Models

For dimensionality reduction techniques such as PCA and Autoencoders, evaluation focuses on information preservation and representation quality rather than explicit clustering accuracy.

1. Explained Variance Ratio (PCA)

$$\text{Explained Variance Ratio} = \frac{\lambda_i}{\sum_{j=1}^{n} \lambda_j}$$

where λ_i are eigenvalues of the covariance matrix.

2. Reconstruction Error

$$\text{Reconstruction Error} = \|X - \hat{X}\|^2$$

Lower reconstruction error indicates better representation learning.

Interpretation

Effective dimensionality reduction retains essential data characteristics while discarding noise.

External and Domain-Based Evaluation

In some practical settings, partial labels or domain expertise may be available to support indirect validation of unsupervised learning outcomes.

- Comparison with known groupings
- Expert interpretation
- Downstream task performance (e.g., improved classification)

Practical Insight

The usefulness of unsupervised learning is often validated by its impact on real-world decision-making.

Overfitting in Unsupervised Learning

Overfitting in unsupervised learning occurs when the model captures noise rather than meaningful structure, resulting in patterns that fail to generalize.

- Excessive number of clusters

- Overly complex latent representations

Goal

The objective is to learn stable and interpretable structures that generalize to new data.

Key Takeaway: Model evaluation in unsupervised learning relies on structural quality, statistical consistency, and practical usefulness rather than direct prediction accuracy.

Chapter 2

Supervised Learning Algorithms

Introduction to Supervised Learning

The Labeled Data Paradigm

Supervised learning is a foundational method where a model learns a mapping from inputs to outputs using *labeled training data*. Each example consists of an input vector $\mathbf{x}$ and a corresponding target value y. The objective is to learn a function $h(\mathbf{x})$ that generalizes to unseen data.

Supervised learning algorithms are the engine behind modern intelligence, powering:

- **Predictive Analytics:** Forecasting market trends and sales.
- **Medical Diagnosis:** Identifying pathologies from clinical imaging.
- **Credit Risk:** Assessing the likelihood of loan default.
- **Recognition Systems:** Processing high-dimensional image and speech signals.

This chapter explores the mathematical foundations, visual interpretations, and practical implementations of these algorithms using `scikit-learn`.

2.1 Linear Regression

Linear Regression models the relationship between a dependent variable and one or more independent variables by fitting a linear equation to ob-

served data. Despite its simplicity, it remains a cornerstone of statistical modeling.

Problem Formulation

Given a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$ where $x_i \in \mathbb{R}^d$ is the feature vector and $y_i \in \mathbb{R}$ is the target, we assume the hypothesis:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_d x_d$$

In compact **vectorized form**, we represent this as:

$$\hat{y} = \boldsymbol{\theta}^T \mathbf{x}$$

where $\boldsymbol{\theta}$ is the parameter vector including the bias term θ_0.

Optimization: Loss Function and Gradient Descent

To find the optimal line, we minimize the **Mean Squared Error (MSE)**, which penalizes larger deviations more heavily:

$$J(\boldsymbol{\theta}) = \frac{1}{2n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

We solve for $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ using **Gradient Descent**. The parameters are updated iteratively in the direction of the steepest descent:

$$\theta_j := \theta_j - \alpha \frac{\partial J}{\partial \theta_j}$$

where α is the **learning rate**, a hyperparameter that controls the step size of each update.

Geometric Interpretation

Linear regression fits a hyperplane in high-dimensional space that minimizes the squared vertical distances (residuals) between observed points and the plane.

Note: While we often visualize this in 2D, the logic remains identical for d dimensions, where the "line" becomes a $(d-1)$-dimensional hyperplane.

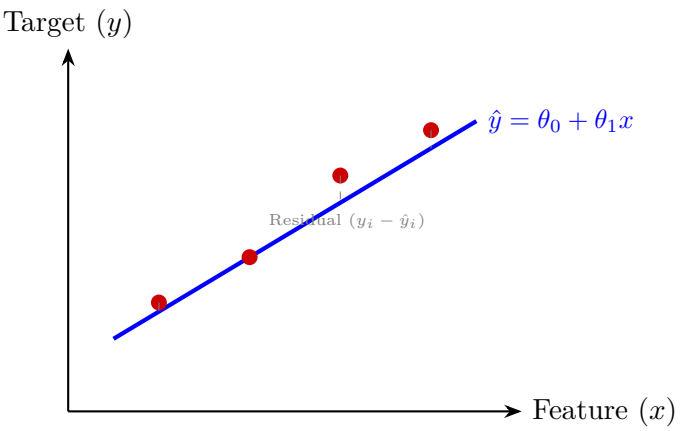


Figure 2.1: Linear Regression: Minimizing the sum of squared residuals.

Example: Predicting House Prices

To understand how the mathematical theory translates into a predictive model, let us consider a simplified real-world scenario: predicting the price of a house based on its size (in 1000s of square feet).

1. The Dataset Suppose we have the following observed data from a neighborhood:

House	Size (x)	Price (y in \$100k)
1	1.0	1.5
2	2.0	2.5
3	3.0	2.8

2. Hypothesis Initialization Assume we start with initial parameters (weights) chosen randomly or set to zero:

- **Bias (θ_0):** 0.0
- **Weight (θ_1):** 0.5

Our hypothesis function is: $\hat{y} = 0.0 + 0.5x$.

3. Calculating Predictions and Error We calculate the prediction ($\hat{y}$) for each house and the resulting error:

- **House 1:** $\hat{y}_1 = 0.5(1.0) = 0.5 \implies$ Error $= (0.5 - 1.5) = -1.0$

- **House 2:** $\hat{y}_2 = 0.5(2.0) = 1.0 \implies \text{Error} = (1.0 - 2.5) = -1.5$
- **House 3:** $\hat{y}_3 = 0.5(3.0) = 1.5 \implies \text{Error} = (1.5 - 2.8) = -1.3$

The **Mean Squared Error (MSE)** for this initial line is:

$$J(\theta) = \frac{1}{2(3)}[(-1.0)^2 + (-1.5)^2 + (-1.3)^2] = \frac{1}{6}[1.0 + 2.25 + 1.69] \approx 0.823$$

4. Parameter Update (One Step of Gradient Descent) Using a learning rate $\alpha = 0.1$, the algorithm calculates the gradient to "tilt" the line toward the data points. After many iterations, the parameters might converge to:

$$\boldsymbol{\theta}^* = [0.95, 0.65]^T$$

5. Final Prediction Now, if a new house comes onto the market with a size of **2.5 (2500 sq. ft)**, we can predict its price:

$$\hat{y}_{new} = 0.95 + 0.65(2.5) = 2.575$$

Predicted Price: \$257,500

Interpretation: Notice that θ_0 (0.95) represents the base price of a house regardless of size, while θ_1 (0.65) represents the increase in price per 1000 square feet. This demonstrates the **interpretability** of Linear Regression.

Advantages and Limitations

The efficacy of Linear Regression is highly dependent on the nature of the data. While it serves as a robust baseline, its mathematical assumptions impose certain constraints on performance.

Advantages	Limitations
Simple and interpretable: Coefficients reveal feature influence clearly.	**Sensitive to outliers:** Extreme values disproportionately affect the fit.
Computationally efficient: Minimal CPU/Memory cost during training.	**Assumes linearity:** Cannot model curved or non-linear data patterns.
Works well for linear trends: Highly accurate for data with constant rates of change.	**Poor for complex patterns:** Limited performance on high-dimensional, intricate data.

Warning: Sensitivity to Outliers

Because the **MSE loss function** squares the residuals, a single data point that is far from the general trend (an outlier) can exert significant "leverage," shifting the entire regression line and reducing the model's accuracy for the majority of the data.

Final Takeaway *Linear Regression is often the first algorithm an AI practitioner should try. If the relationship between features and the target is roughly linear, it provides a highly interpretable and efficient solution that is difficult to beat with more complex models.*

2.2 Logistic Regression

➢ Classification via Regression

Despite its name, **Logistic Regression** is a fundamental **classification algorithm**. It is the go-to method for binary classification tasks where the goal is to predict the probability that a given input belongs to a specific category (e.g., Spam vs. Not Spam).

Motivation: Why Not Linear Regression?

Linear regression predicts continuous values in the range $(-\infty, \infty)$. For classification, this is problematic because:

- Outliers can drastically shift the decision boundary.
- Predictions can be greater than 1 or less than 0, which lack probabilistic meaning.

Logistic Regression solves this by "squashing" the linear output through a non-linear activation function.

The Sigmoid (Logistic) Function

The heart of this model is the **Sigmoid function**, $\sigma(z)$, which maps any real-valued number to a value between 0 and 1:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.1}$$

As $z \to \infty$, $\sigma(z) \to 1$. As $z \to -\infty$, $\sigma(z) \to 0$. When $z = 0$, $\sigma(z) = 0.5$.

Model Hypothesis and Decision Boundary

We define the probability that an input $\mathbf{x}$ belongs to class 1 ($y = 1$) as:

$$P(y = 1 \mid \mathbf{x}; \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}^T \mathbf{x}) = \frac{1}{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}}} \tag{2.2}$$

To make a discrete prediction $\hat{y} \in \{0, 1\}$, we apply a **threshold** (typically 0.5):

$$\hat{y} = \begin{cases} 1 & \text{if } \sigma(\boldsymbol{\theta}^T \mathbf{x}) \geq 0.5 \implies (\boldsymbol{\theta}^T \mathbf{x} \geq 0) \\ 0 & \text{otherwise} \end{cases}$$

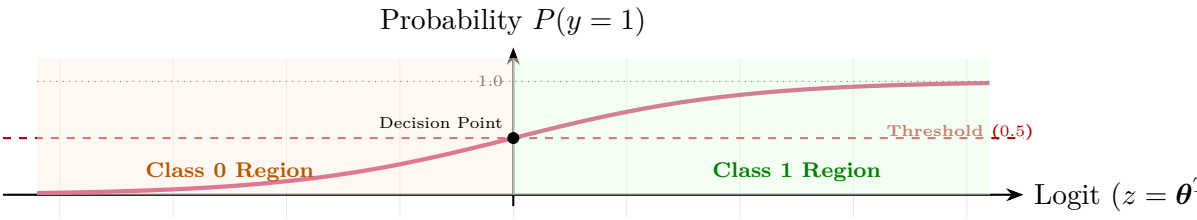


Figure 2.2: The Sigmoid activation function mapping linear combinations to probabilities.

The Loss Function: Binary Cross-Entropy (Log Loss)

In Linear Regression, we used Mean Squared Error (MSE). However, if we apply MSE to the non-linear Sigmoid output, the resulting cost function becomes **non-convex**. This means the "valley" has many ripples (local minima), making it nearly impossible for Gradient Descent to find the true global minimum.

To ensure a smooth, **convex** optimization surface, we use **Binary Cross-Entropy (BCE)**.

1. The Mathematical Definition The average loss over n training examples is defined as:

$$J(\boldsymbol{\theta}) = -\frac{1}{n} \sum_{i=1}^{n} \underbrace{[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]}_{\text{Cost for a single example}}$$

By: Engr. Dr. Muhammad Siddique, Postdoc AI (YALE University, USA).
HOD Artificial Intelligence, NFC IET Multan, msiddique@nfciet.edu.pk

2. The "Switch" Logic This formula acts like a logical switch. Since y_i is always either 0 or 1, one part of the equation always disappears:

- **Case 1: The Actual Label is $y_i = 1$** The second term $(1 - 1) = 0$ vanishes. We are left with $-\log(\hat{y}_i)$.
  - If the model predicts $\hat{y} = 0.99$ (Correct), $-\log(0.99) \approx 0.01 \rightarrow$ **Tiny Penalty**.
  - If the model predicts $\hat{y} = 0.01$ (Wrong), $-\log(0.01) \approx 4.60 \rightarrow$ **Huge Penalty**.
- **Case 2: The Actual Label is $y_i = 0$** The first term vanishes. We are left with $-\log(1 - \hat{y}_i)$.
  - If the model predicts $\hat{y} = 0.01$ (Correct), $-\log(0.99) \approx 0.01 \rightarrow$ **Tiny Penalty**.
  - If the model predicts $\hat{y} = 0.99$ (Wrong), $-\log(0.01) \approx 4.60 \rightarrow$ **Huge Penalty**.

3. Why the "Log"? The use of the logarithm ensures that the penalty grows **exponentially** as the prediction deviates from the actual label. This "punishes" the model severely for being confident and wrong (e.g., predicting 0.99 probability for a class 0 instance), which forces the model to learn much faster.

Note: BCE measures the "dissimilarity" between the predicted probability distribution and the actual distribution. In AI, we call this the **Kullback–Leibler (KL) Divergence**.

Numerical Trace: The Heart Disease Prediction Pipeline

Let us trace how the model processes a specific patient's data. This "end-to-end" look reveals how a raw medical measurement is transformed into a probabilistic risk assessment.

1. Scenario Setup & Parameters

We predict the probability (P) of heart disease ($y = 1$) based on **Serum Cholesterol** (x_1). Through optimization, our model learned these "knowledge" parameters:

- **Bias ($\theta_0 = -5.0$):** The baseline log-odds (negative means low default risk).
- **Weight ($\theta_1 = 0.02$):** The "impact factor" per mg/dL of cholesterol.

2. Step-by-Step Execution for Patient A ($x_1 = 200$)

Step 1: The Linear Logit (z)

Combine the feature with the weights to get the raw score:

$$z = \theta_0 + \theta_1(x_1)$$

$$z = -5.0 + 0.02(200) = -5.0 + 4.0 = \mathbf{-1.0}$$

A negative logit suggests the probability will be less than 50%.

Step 2: The Sigmoid "Squash"

Map the raw score to a probability space $[0, 1]$ using the sigmoid function:

$$P(y = 1) = \sigma(z) = \frac{1}{1 + e^{-(\mathbf{-1.0})}} = \frac{1}{1 + 2.718} \approx \mathbf{0.269}$$

The Risk Factor is 26.9%.

Step 3: The Classification Decision

Compare the probability to our **Decision Threshold** ($\tau = 0.5$):

$$\text{Prediction } \hat{y} = \begin{cases} 1 & \text{if } P \geq 0.5 \\ 0 & \text{if } P < 0.5 \end{cases} \implies \hat{\mathbf{y}} = \mathbf{0} \text{ (Negative)}$$

3. Sensitivity Analysis: The Impact of Rising Cholesterol

What happens if the patient's cholesterol rises to 300?

- **New Logit:** $z = -5.0 + 0.02(300) = \mathbf{+1.0}$
- **New Probability:** $\sigma(+1.0) \approx \mathbf{0.731}$ (Risk jumps to **73.1%**).

At this point, the prediction flips to **Positive** ($y = 1$).

4. Strategic Interpretability

☞ **Finding the "Tipping Point" (Decision Boundary)**
To find exactly when a patient is considered "High Risk," we set $z = 0$:

$$-5 + 0.02x_1 = 0 \implies \mathbf{x_1 = 250}\ \text{mg/dL}$$

In this AI model, **250 mg/dL** is the critical threshold separating healthy from at-risk predictions.

Interpretability Note (The Odds Ratio): If you want to know how much riskier each mg/dL is, we look at e^{θ_1}. $e^{0.02} \approx 1.02$. This means for every 1 unit increase in cholesterol, the **odds** of heart disease increase by **2%**.

2.3 k-Nearest Neighbors (k-NN)

➢ **Instance-Based Lazy Learning**

k-Nearest Neighbors (k-NN) is a classical yet powerful **non-parametric** learning algorithm that makes no prior assumptions about the underlying data distribution. Unlike parametric models, k-NN does not learn an explicit mapping function during training. Instead, it stores the entire dataset and defers all learning to the **prediction (inference) phase**. For this reason, k-NN is widely referred to as a **lazy learning** algorithm.

1. Workflow of the k-NN Algorithm

Although k-NN avoids a traditional training phase, it follows a well-defined and systematic pipeline during inference. The computational responsibility is entirely shifted to prediction time, where the algorithm performs three sequential operations: comparison, selection, and aggregation.

➤ Computational Perspective: Why k-NN is Expensive

For AI practitioners, understanding the computational behavior of k-NN is essential. The inference-time complexity of k-NN is dominated by:

$$\mathcal{O}(n \cdot d)$$

- n = **Number of Samples:** Each query point must be compared against *every* stored instance in the dataset.
- d = **Feature Dimension:** For each comparison, a distance metric is computed across all features.

As a result, k-NN scales poorly with large datasets and high-dimensional feature spaces.

2. Neighbor Identification All distances are sorted to extract the nearest neighbors:

$$\text{Sorting Cost} \approx \mathcal{O}(n \log n)$$

Scalability Warning: If the dataset grows to **1 million samples** with **1,000 features**, a single prediction may require **one billion operations**. Unlike neural networks=where inference is fast after training=k-NN becomes increasingly expensive as data volume increases.

Counting the Operations:

Consider a practical deployment scenario in a small medical diagnostic system.

- **Dataset Size (n):** 1,000 historical patient records
- **Feature Space (d):** 10 clinical attributes (such as Age, BMI, Blood Pressure, etc.)

When a **new patient** arrives, the k-Nearest Neighbors (k-NN) algorithm performs the following computations:

1. Distance Evaluation The algorithm computes the Euclidean distance between the new patient and *every* existing patient record. Each distance calculation uses all available features.

$$1{,}000 \times 10 = \mathbf{10{,}000}$$

This means that **10,000 feature-wise operations** are required for a single prediction. Each operation typically involves subtraction, squaring, and addition.

2. Neighbor Identification (Sorting) After computing all distances, the algorithm sorts them in ascending order to identify the nearest neighbors.

The sorting complexity is approximately:

$$\mathcal{O}(n \log n)$$

For $n = 1{,}000$:

$$n \log_2 n = 1{,}000 \times \log_2(1{,}000) \approx 1{,}000 \times 10 = \mathbf{10{,}000}$$

Therefore, around **10,000 sorting operations** are needed to arrange the distances before selecting the nearest neighbors.

As the number of samples or features increases, both these costs grow rapidly. This explains why k-NN becomes computationally expensive and slow when applied to large datasets or high-dimensional data.

Operational Phases of k-NN

➔ Phase I: Distance Computation

For a given unlabeled query point $\mathbf{x}_q$, the algorithm computes its dissimilarity with every stored instance $\mathbf{x}_i$.

$$d(\mathbf{x}_q, \mathbf{x}_i) = \sqrt{\sum_{j=1}^{d} (\mathbf{x}_{q,j} - \mathbf{x}_{i,j})^2}$$

Key Insight: This phase dominates computational cost and justifies the classification of k-NN as a *lazy learner*.

➔ Phase II: Neighbor Selection

Distances are sorted, and the closest k instances are selected.

- **Small k:** Captures local patterns but is sensitive to noise (high variance).
- **Large k:** Produces smoother decision boundaries but may miss fine-grained structure (high bias).

➔ Phase III: Decision Aggregation

The final prediction is derived by aggregating the labels of the selected neighbors.

- **Classification:**

$$\hat{y} = \text{mode}(y_1, y_2, \ldots, y_k)$$

- **Regression:**

$$\hat{y} = \frac{1}{k} \sum_{i=1}^{k} y_i$$

Best Practice: Use an **odd** value of k in binary classification to prevent tie votes.

Geometric Interpretation of Neighborhood Search

The core logic of k-NN lies in its local geometry. Figure 2.3 visualizes how the classification of a query point is not fixed, but rather dynamic, shifting as the neighborhood radius expands to include more neighbors.

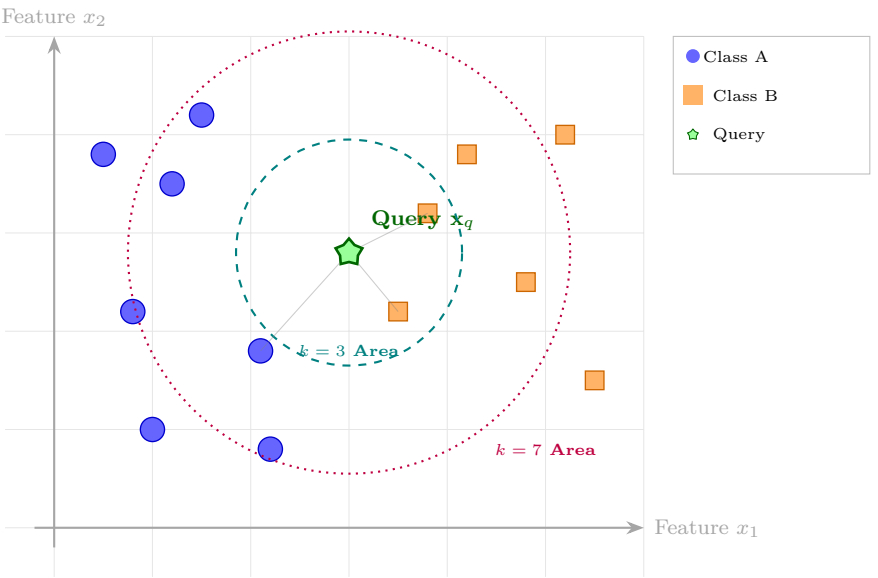


Figure 2.3: Geometric intuition of k-NN: expanding the neighborhood radius alters the majority vote and hence the predicted class.

Analysis of Figure 2.3

- **Case $k = 3$:** The nearest neighbors are **2 Squares** and **1 Circle**. By majority rule, the model predicts **Class B**.
- **Case $k = 7$:** The search radius expands. It now captures more global trends, encompassing **4 Circles** and **3 Squares**. The prediction flips to **Class A**.

This highlights the role of k as a **smoothing hyperparameter**. Small k values make the model sensitive to local noise, whereas large k values make the model more robust but potentially less precise.

Counting the Operations

Consider a practical deployment scenario in a small medical diagnostic system:

- **Dataset Size (n):** 1,000 historical patient records
- **Feature Space (d):** 10 clinical attributes (Age, BMI, Blood Pressure, etc.)

When a **new patient** arrives, the algorithm performs the following computations:

1. Distance Evaluation The Euclidean distance between the new patient and every existing record is calculated:

$$1{,}000 \times 10 = \mathbf{10{,}000} \text{ feature-wise operations}$$

Each operation involves subtraction, squaring, and accumulation.

2. Neighbor Identification All distances are sorted to extract the nearest neighbors:

$$\text{Sorting Cost} \approx \mathcal{O}(n \log n)$$

Scalability Warning: If the dataset grows to **1 million samples** with **1,000 features**, a single prediction may require **one billion operations**. Unlike neural networks=where inference is fast after training=k-NN becomes increasingly expensive as data volume increases.

Example: Fruit Classification

Imagine we are classifying fruits based on two features: **Sweetness** (x_1) and **Crunchiness** (x_2). We have a dataset of 4 fruits:

Sample	Sweetness (x_1)	Crunchiness (x_2)	Label (y)
1	7	7	Apple
2	8	6	Apple
3	3	4	Orange
4	1	2	Orange

The Query: A new fruit arrives with **Sweetness = 4** and **Crunchiness = 3**. We want to classify it using $k = 3$.

1. **Calculate Euclidean Distances to Query $Q(4, 3)$:**
   - $d(Q, 1) = \sqrt{(4-7)^2 + (3-7)^2} = \sqrt{9+16} = \mathbf{5.0}$
   - $d(Q, 2) = \sqrt{(4-8)^2 + (3-6)^2} = \sqrt{16+9} = \mathbf{5.0}$
   - $d(Q, 3) = \sqrt{(4-3)^2 + (3-4)^2} = \sqrt{1+1} = \mathbf{1.41}$ (Rank 1)
   - $d(Q, 4) = \sqrt{(4-1)^2 + (3-2)^2} = \sqrt{9+1} = \mathbf{3.16}$ (Rank 2)
2. **Identify 3 Nearest Neighbors:** Samples 3, 4, and (1 or 2). Let's pick Sample 1 as the third.
3. **Majority Vote:**
   - Samples 3 & 4 are **Oranges**. Sample 1 is an **Apple**.
   - **Result:** 2 votes for Orange vs 1 vote for Apple.

Final Prediction: The new fruit is classified as an **Orange**.

Visualizing the Decision Boundary

The Bias–Variance Tradeoff in k-NN

The choice of k is critical and directly influences the model's complexity:

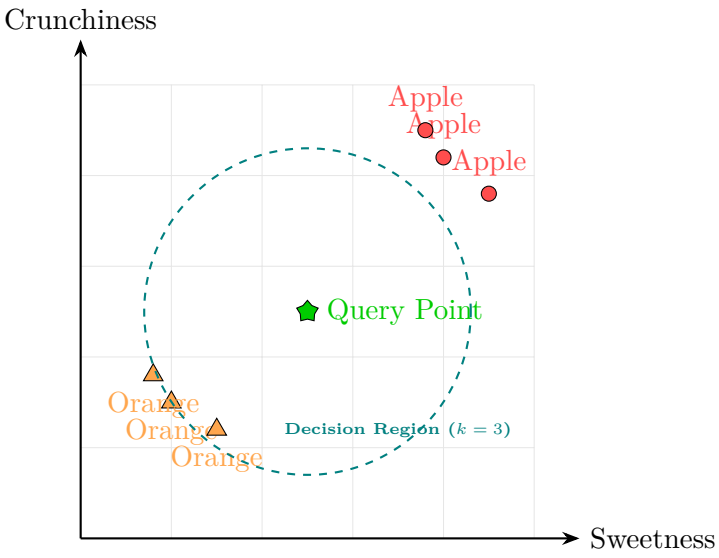


Figure 2.4: Geometric representation of k-NN. The query point's label is determined by the majority of neighbors within the circular boundary.

Hyperparamet	Model Characteristic	Visual Effect
Small k (e.g., $k = 1$)	**Low Bias / High Variance:** Captures fine details but is sensitive to noise/outliers.	Jagged, complex decision boundaries.
Large k (e.g., $k = 100$)	**High Bias / Low Variance:** Models the general trend but ignores local patterns.	Smooth, simple decision boundaries.

Note: Always **scale your features** (e.g., Normalization) before using k-NN. Because it relies on distance, a feature with a larger range (like Salary) will dominate a feature with a smaller range (like Age) if not scaled.

2.4 Support Vector Machines (SVM)

Support Vector Machines: Learning by Geometry

Support Vector Machines (SVMs) are supervised learning models that approach classification from a **geometric and optimization-driven perspective**. Rather than merely separating data points, an SVM seeks a boundary that is **maximally distant from the data of both classes**. This idea of **maximum separation** gives SVMs their remarkable robustness and generalization ability.

Geometric Illustration: Why Some Separations Are Better

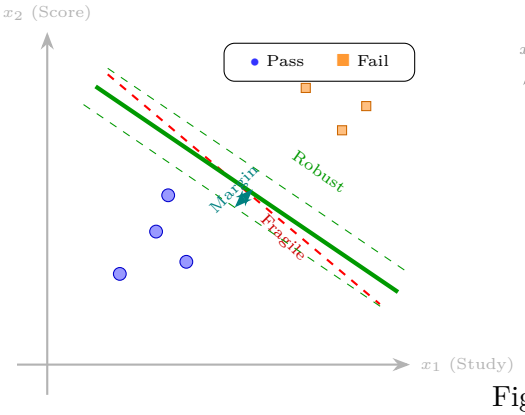


Figure 2.5: Robust vs. Fragile separators based on margin width.

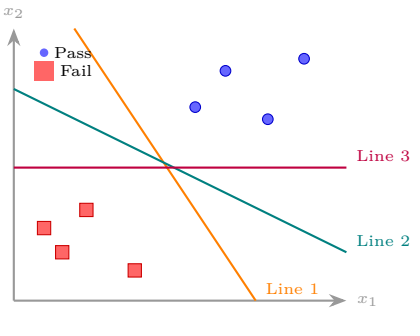


Figure 2.6: Non-uniqueness: Multiple valid separation lines.

1. Beyond Simple Separation: The Search for Robustness

In a typical binary classification task, such as predicting if a student will **Pass (+1)** or **Fail (−1)** based on **Study Hours** (x_1) and **Internal Scores** (x_2), we often find that a straight line can separate the data.

The SVM Principle: We must choose the boundary that maximizes the **Margin**=the "buffer zone" between classes. A line passing too close to a data point (like Line 1 or 3 in Figure 2.6) is sensitive to noise; Line 2 is more robust because it maintains a safer distance from both clusters.

This idea can be understood geometrically by comparing multiple separating hyperplanes. Although all may correctly classify the training data, only one achieves the **maximum margin**, making it the optimal SVM solution.

Normal Distance of Points from a 2D Plane (Line)

Generic Formula
For a line (2D plane)

$$ax + by + c = 0$$

and a point

$$P(x_0, y_0),$$

the normal (perpendicular) distance from the point to the line is

$$\boxed{d = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}}$$

We consider the line

$$2x + y - 4 = 0$$

and two different points.

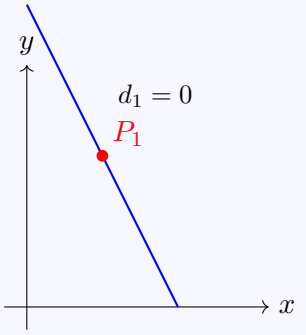
Case 1: Point on the Plane
Point:

$$P_1(1, 2)$$

Distance:

$$d_1 = \frac{|2(1) + 1(2) - 4|}{\sqrt{5}} = 0$$

Interpretation: The point lies exactly on the plane.



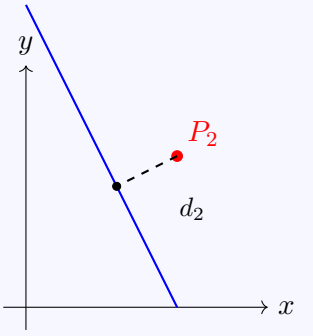
Case 2: Point Outside the Plane
Point:

$$P_2(2, 2)$$

Distance:

$$d_2 = \frac{|2(2) + 1(2) - 4|}{\sqrt{5}} = \frac{2}{\sqrt{5}}$$

Interpretation: The point lies outside the plane and has a non-zero normal distance.



Summary
- A point on the plane has **zero normal distance**. - A point outside the plane has a **positive normal distance**, equal to the length of the perpendicular from the point to the plane.

2.5 Hyperplane - A Deep Geometric Explanation

The term *hyperplane* may initially appear abstract or intimidating. However, it naturally arises when we study how geometric objects that **separate space** generalize as the dimensionality increases.

2.5.1 Separating Objects Across Dimensions

To understand hyperplanes, let us examine how separation works in spaces of increasing dimension.

- In **one-dimensional space** ($\mathbb{R}^1$), a **point** separates a line into two half-lines.
- In **two-dimensional space** ($\mathbb{R}^2$), a **line** separates the plane into two half-planes.
- In **three-dimensional space** ($\mathbb{R}^3$), a **plane** separates space into two half-spaces.
- In n**-dimensional space** ($\mathbb{R}^n$), the separator is an **(n–1)-dimensional object** called a **hyperplane**.

Core Intuition

In any space, the object that separates the space into two regions always has **one dimension less** than the space itself. Such an object is called a **hyperplane**.

Fundamental Pattern

Dimension of separator = Dimension of space -1

By: Engr. Dr. Muhammad Siddique, Postdoc AI (YALE University, USA).
HOD Artificial Intelligence, NFC IET Multan, msiddique@nfciet.edu.pk

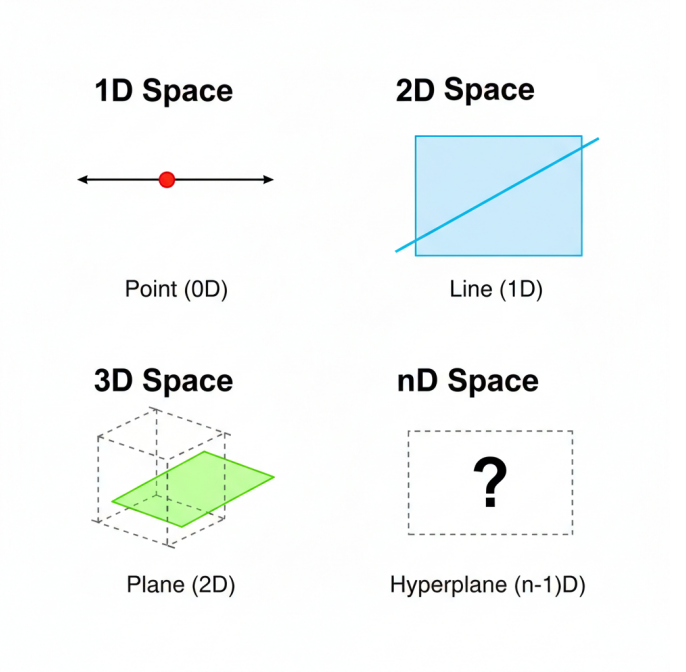


Figure 2.7: Evolution of Separating Boundaries Across Dimensions

2.5.2 Formal Mathematical Definition of hyperplane

In an n-dimensional real vector space $\mathbb{R}^n$, a hyperplane is defined as the set of all points $\mathbf{x}$ satisfying:

$$\langle \mathbf{w}, \mathbf{x} \rangle + b = 0$$

where:

- $\mathbf{w} \in \mathbb{R}^n$ is the **normal vector**
- $b \in \mathbb{R}$ is the **bias (offset)**

Mathematical Consequence

This equation always defines an **(n–1)-dimensional flat (imaginary) surface** embedded in an n-dimensional space.

2.5.3 Why the Prefix *"Hyper"*?

The prefix *hyper-* originates from the Greek language and means **"beyond"**.

Meaning of "Hyper"

A hyperplane is a *plane beyond three dimensions*, whose direct visualization is no longer possible.
In dimensions greater than three, the separating object is no longer a plane in the usual sense, yet it behaves mathematically exactly like a plane. Hence, it is called a **hyperplane**.

2.5.4 Role of Hyperplanes in Support Vector Machines

In Support Vector Machines:

- Each data point is represented as a vector in $\mathbb{R}^n$
- Each feature corresponds to one dimension
- The classifier must work independently of dimension

Why Hyperplanes Are Ideal for SVM

Hyperplanes provide a linear, dimension-independent, and optimizable decision boundary suitable for both low- and high-dimensional spaces.

A data point is classified according to the sign of:

$$\langle \mathbf{w}, \mathbf{x} \rangle + b$$

which indicates on which side of the hyperplane the point lies.

A hyperplane is called a hyperplane because it generalizes the concept of a line and a plane to an $(n-1)$-dimensional separating surface in n-dimensional space.

Geometric Illustration of Hyperplanes and Margin

Key Takeaway: Even if multiple hyperplanes correctly separate the training data, SVM prefers the one that is **farthest from the nearest data points**. This geometric preference is what gives SVM its strong resistance to noise and its excellent generalization performance.

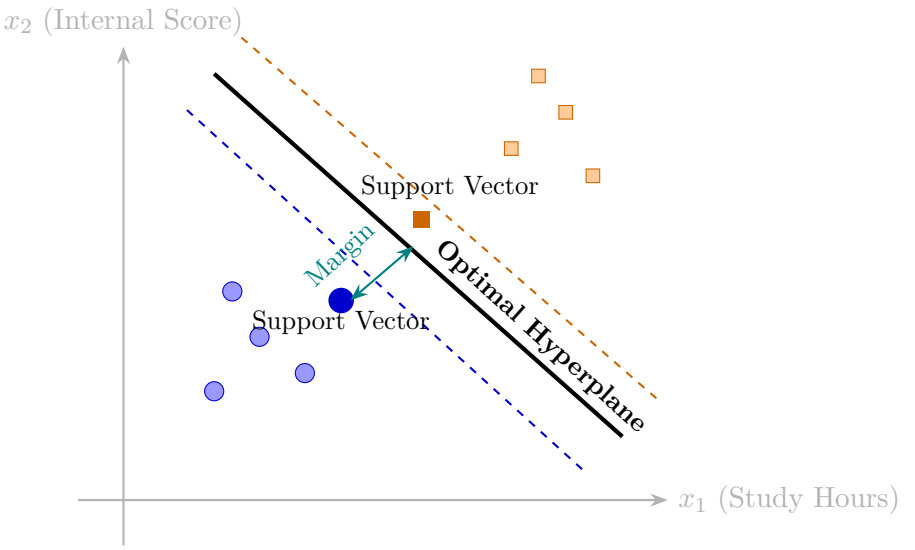


Figure 2.8: Among many possible separating lines, SVM selects the hyperplane that maximizes the margin. The closest points=called support vectors=determine the position of the hyperplane.

Example: Hyperplane with Multiple Points

Assume an SVM trained on two features learns the parameters:
Hwere the values of W and b are initial guess.

$$\mathbf{w} = \begin{bmatrix} 1.5 \\ -2 \end{bmatrix}, \qquad b = -1$$

The decision boundary is:

$$1.5x_1 - 2x_2 - 1 = 0 \quad \Rightarrow \quad x_2 = 0.75x_1 - 0.5$$

Now consider three data points:

$$\mathbf{x}^{(1)} = (2, 1), \quad \mathbf{x}^{(2)} = (3, 2.5), \quad \mathbf{x}^{(3)} = (1, 2)$$

Step 1: Compute Decision Scores

$$f(\mathbf{x}^{(1)}) = 1.5(2) - 2(1) - 1 = \mathbf{0}$$
$$f(\mathbf{x}^{(2)}) = 1.5(3) - 2(2.5) - 1 = \mathbf{-1.5}$$
$$f(\mathbf{x}^{(3)}) = 1.5(1) - 2(2) - 1 = \mathbf{-3.5}$$

Interpretation:

- $\mathbf{x}^{(1)}$ lies **exactly on the hyperplane**
- $\mathbf{x}^{(2)}$ lies on the **negative side, close to margin**
- $\mathbf{x}^{(3)}$ lies **far from the boundary** (high confidence)

Distance and Margin Analysis

The perpendicular distance of a point from the hyperplane is:

$$d(\mathbf{x}) = \frac{|\mathbf{w}^T\mathbf{x} + b|}{\|\mathbf{w}\|}$$

First compute the norm:

$$\|\mathbf{w}\| = \sqrt{1.5^2 + (-2)^2} = \sqrt{6.25} = 2.5$$

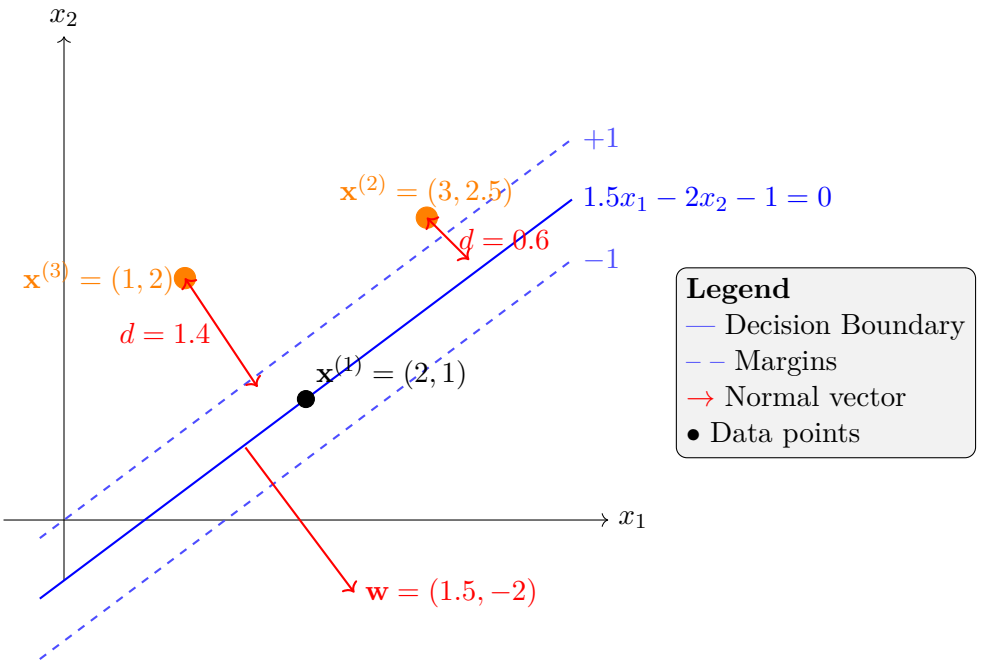


Figure 2.9: Geometric interpretation of the numerical SVM example: hyperplane, margin, normal vector, and distances.

Distances from the Hyperplane

$$d(\mathbf{x}^{(2)}) = \frac{1.5}{2.5} = \mathbf{0.6}$$
$$d(\mathbf{x}^{(3)}) = \frac{3.5}{2.5} = \mathbf{1.4}$$

Margin Geometry: The total margin is: $0.6 + 1.4 = 2$

The total margin width is:

$$\text{Margin Width} = \frac{2}{\|\mathbf{w}\|} = \frac{2}{2.5} = \mathbf{0.8}$$

Points with distance ≤ 0.4 lie **inside the margin**.

Optimization Perspective

Why SVM Minimizes $\|\mathbf{w}\|^2$:

- Smaller $\|\mathbf{w}\| \Rightarrow$ Wider margin
- Wider margin $\Rightarrow$ Better resistance to noise
- Better resistance $\Rightarrow$ Improved generalization

The Geometry of Optimization: Why Minimize $\|\mathbf{w}\|^2$?

The core objective of an SVM is to find the **Maximum Margin Hyperplane**. Mathematically, this leads us to an inverse relationship between the weight vector $\mathbf{w}$ and the margin width.

- **Creating a "Safety Buffer":**
  - A smaller $\|\mathbf{w}\|$ forces the decision boundary to be as far as possible from the nearest training points (Support Vectors).
  - This creates a "no-man's land" where no data exists, ensuring the model isn't "suffocating" the data points.
- **Robustness to Noise:**
  - In real-world AI, data is messy. If the margin is thin, a slight measurement error in a test data point could push it across the boundary, causing a misclassification.
  - A wider margin (small $\|\mathbf{w}\|$) provides a buffer that absorbs noise and prevents small fluctuations from changing the prediction.
- **Combating Overfitting (Generalization):**
  - By minimizing $\|\mathbf{w}\|^2$, we are effectively applying **Regularization**.
  - It prevents the model from becoming too complex or "wiggly" just to fit a single outlier.
  - *Insight:* A simpler, wider boundary usually performs better on "unseen" future data than a complex, tight boundary.

The Geometry of Optimization: Why Minimize $\|\mathbf{w}\|^2$?

Note Small $\|\mathbf{w}\|$ → Large Gap → High Stability → Reliable AI.

Thus, the geometric objective becomes:

$$\min_{\mathbf{w},b} \quad \frac{1}{2}\|\mathbf{w}\|^2$$

Final Insight: SVM classification is not driven by all points equally. Only the points closest to the hyperplane=**support vectors**=define the margin and ultimately control the decision boundary.

Final Insight: Moving non-support-vector points does not affect the classifier. Only support vectors control the position of the hyperplane.

Example: Spam detection

Let us now trace the complete SVM decision process numerically.

Email Classification Example

Task: Spam detection
Features:

- x_1: Number of suspicious keywords
- x_2: Number of external links

Model Parameters:

$$\mathbf{w} = \begin{bmatrix} 0.8 \\ 0.6 \end{bmatrix}, \quad b = -3$$

New Email:

$$x_1 = 4, \quad x_2 = 3$$

Decision Function Evaluation:

$$\begin{aligned} f(\mathbf{x}) &= \mathbf{w}^T\mathbf{x} + b \\ &= (0.8)(4) + (0.6)(3) - 3 \\ &= 3.2 + 1.8 - 3 \\ &= \mathbf{2.0} \end{aligned}$$

Prediction:
Since $f(\mathbf{x}) > 0$, the email is classified as **Spam (+1)**.

Confidence Interpretation:
Because $|2.0| > 1$, the email lies outside the margin, indicating strong confidence.

```
# Example: Spam detection using SVM
# Feature vector of the new email
# x1: number of suspicious keywords
# x2: number of external links
```

```
x = [4, 3]

# Model parameters
w = [0.8, 0.6]    # weight vector
b = -3            # bias term

# Compute decision function f(x) = w^T x + b
decision_value = w[0] * x[0] + w[1] * x[1] + b

print("Decision function value f(x):", decision_value)

# Classification based on sign of f(x)
if decision_value > 0:
    prediction = "Spam (+1)"
else:
    prediction = "Not Spam (-1)"

print("Prediction:", prediction)

# Confidence interpretation based on margin
if abs(decision_value) > 1:
    confidence = "High confidence (outside the margin)"
else:
    confidence = "Low confidence (inside the margin)"

print("Confidence:", confidence)
```

Listing 2.1: Python implementation of SVM spam detection example

2.6 The Kernel Trick: From 2D to 3D Visualization

Real-world datasets are rarely perfectly separable. Noise, overlapping classes, and measurement errors make strict decision boundaries undesirable. Support Vector Machines overcome these challenges through two powerful mechanisms: **soft margins** and the **kernel trick**. This section presents a complete mathematical, geometric, and visual explanation of how nonlinear data in two dimensions becomes linearly separable in higher-dimensional feature spaces.

2.6.1 Soft-Margin Support Vector Machine

The hard-margin SVM enforces strict separation, making it highly sensitive to noise. The soft-margin SVM relaxes this constraint by introducing **slack variables** ξ_i, which allow controlled violations of the margin.

$$\min_{\mathbf{w},b,\boldsymbol{\xi}} \quad \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i$$

subject to

$$y_i(\mathbf{w}^\top \mathbf{x}_i + b) \geq 1 - \xi_i, \qquad \xi_i \geq 0$$

Interpretation of Slack Variables and C

- $\xi_i = 0$: correctly classified and outside the margin
- $0 < \xi_i < 1$: inside the margin but correctly classified
- $\xi_i \geq 1$: misclassified point
- Large C: narrow margin, low tolerance, risk of overfitting
- Small C: wider margin, higher tolerance, better generalization

2.6.2 Nonlinear Data in the Input Space (2D)

Consider the following dataset in two-dimensional input space. The red class lies near the origin, while the blue class surrounds it, forming a nonlinearly separable structure.

Table 2.1: 2D Input Space Data Points

Point	x_1	x_2	Class
A	0.2	0.1	+1 (Red)
B	-0.1	-0.3	+1 (Red)
C	0.0	0.4	+1 (Red)
D	1.5	1.2	-1 (Blue)
E	-1.4	1.5	-1 (Blue)
F	1.1	-1.3	-1 (Blue)
G	-1.2	-1.2	-1 (Blue)

Table 2.2: Mapped Feature Space Values (3D)

Point	x_1	x_2	$z = x_1^2 + x_2^2$
A	0.2	0.1	0.05
B	-0.1	-0.3	0.10
C	0.0	0.4	0.16
D	1.5	1.2	3.69
E	-1.4	1.5	4.21
F	1.1	-1.3	2.90
G	-1.2	-1.2	2.88

In this space, no straight line can separate the two classes. **Kernel Function $\Phi(\mathbf{x})$ can convert the 2D data to 3D, i.e., when data is input to the Kernel function, it is transformed to separable form. So, it becomes possible to separate the data using Hyperplane.**

How the Kernel Function Measures Similarity (with Example): In the kernel function, $\mathbf{x}$ is the data point that we want to classify, while $\mathbf{z}$ is a data point from the training set. The kernel function measures similarity

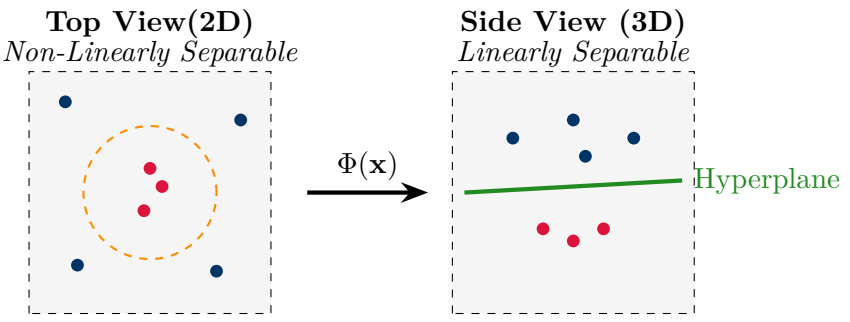


Figure 2.10: Kernel trick: nonlinear circular separation in 2D becomes a linear hyperplane in 3D

by computing the distance between $\mathbf{x}$ and $\mathbf{z}$. For the Radial Basis Function (RBF) kernel, similarity is defined as

$$K(\mathbf{x}, \mathbf{z}) = \exp\left(-\gamma \|\mathbf{x} - \mathbf{z}\|^2\right).$$

First, the squared distance $\|\mathbf{x} - \mathbf{z}\|^2$ is calculated. For example, if $\mathbf{x} = (1, 2)$ and $\mathbf{z} = (1, 3)$, then

$$\|\mathbf{x} - \mathbf{z}\|^2 = (1 - 1)^2 + (2 - 3)^2 = 1.$$

Assuming $\gamma = 0.5$, The parameter γ controls how fast similarity decreases as the distance between points increases. Now the kernel value becomes

$$K(\mathbf{x}, \mathbf{z}) = \exp(-0.5 \times 1) \approx 0.61.$$

This value is relatively high, which means that $\mathbf{x}$ and $\mathbf{z}$ are similar. If the distance were larger, the kernel value would be much smaller, indicating low similarity. In this way, the kernel function assigns high similarity to nearby points and low similarity to distant points, and the most influential training points are called **support vectors**.

Red points remain near the base of the feature space, while blue points are lifted upward, enabling linear separation.

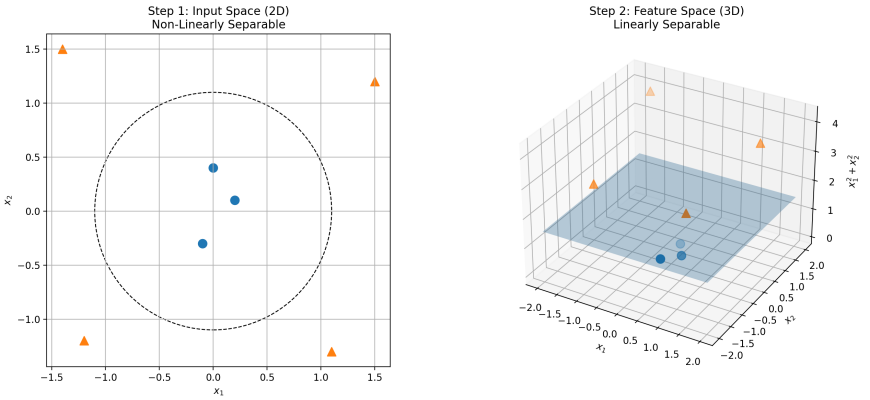


Figure 2.11: Step-by-step illustration of the kernel trick: nonlinear separation in 2D becomes linear in 3D

Step-by-Step Summary of the Transformation

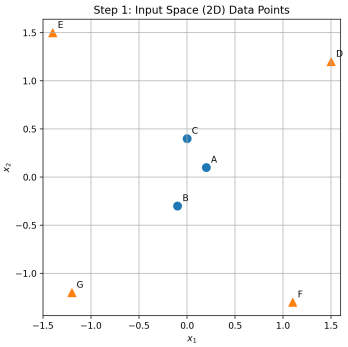
From 2D to 3D: Complete Process

1. Original data is nonlinearly separable in 2D input space.
2. A kernel function implicitly maps data to higher dimensions.
3. Radial distance becomes a new discriminative feature.
4. Points near the center remain low in feature space.
5. Distant points are lifted upward.
6. A linear hyperplane separates the transformed data.

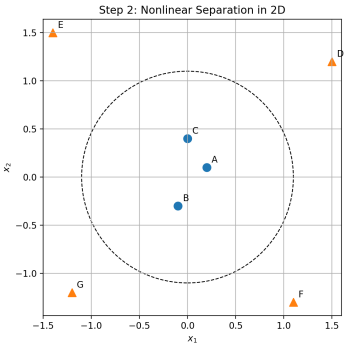
Key Insight: The kernel trick enables SVMs to learn nonlinear decision boundaries by constructing linear separators in high-dimensional feature spaces, without explicitly computing the transformation.

2.6.3 Soft Margin SVM & The Kernel Trick

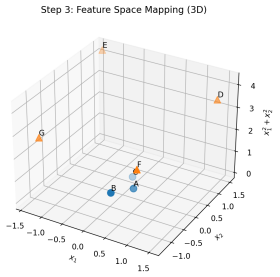
In real-world AI, data is "messy." We use the **Soft Margin** to handle noise and the **Kernel Trick** to handle non-linear patterns.



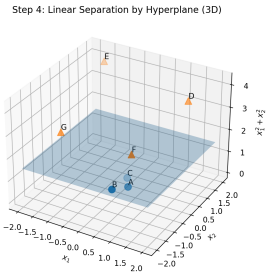
(a) Original data points in 2D input space



(b) Nonlinear decision boundary in 2D



(c) Data mapped into 3D feature space



(d) Linear hyperplane separating data in 3D

Figure 2.12: Step-by-step illustration of nonlinear data separation using feature space transformation

1. Soft Margin: Handling the Noise

The Problem: Hard margins fail if just one outlier is in the wrong place. **The Solution:** We introduce "Slack Variables" (ξ_i) that allow points to be on the "wrong side" for a price.

Objective Function:

$$\min_{\mathbf{w},b,\xi} \underbrace{\frac{1}{2}\|\mathbf{w}\|^2}_{\text{Maximize Margin}} + \underbrace{C\sum_{i=1}^{n}\xi_i}_{\text{Minimize Violations}}$$

1. Soft Margin: Handling the Noise

The "C" Hyperparameter (The Trade-off):

- **Small C:** "Chill" mode. Ignores more errors for a wider margin (Prevents Overfitting).
- **Large C:** "Strict" mode. Penalizes every error heavily, leading to a narrow margin (Risk of Overfitting).

The "C" Hyperparameter (The Trade-off):

- **Small C:** "Chill" mode. Ignores more errors for a wider margin (Prevents Overfitting).
- **Large C:** "Strict" mode. Penalizes every error heavily, leading to a narrow margin (Risk of Overfitting).

Kernel Type	Formula	When to use?
Linear	$K(\mathbf{x}, \mathbf{z}) = \mathbf{x}^\top \mathbf{z}$	High feature count (e.g., Text)
Polynomial	$(\gamma \mathbf{x}^\top \mathbf{z} + r)^d$	Image processing/Specialized geometry
RBF (Gaussian)	$e^{-\gamma \|\mathbf{x}-\mathbf{z}\|^2}$	**The Default.** For complex non-linear data

Summarys:

- **Soft Margin (C)** handles *Outliers*.
- **Kernel Trick (K)** handles *Curvature*.

2.7 Decision Trees

➤ Core Idea: Recursive Partitioning

Decision Trees are non-parametric supervised learning models. Instead of fitting one global equation (like linear regression), a decision tree learns by **repeatedly asking simple questions** about the features.

Each question **splits the data** into smaller and purer groups until predictions become easy.

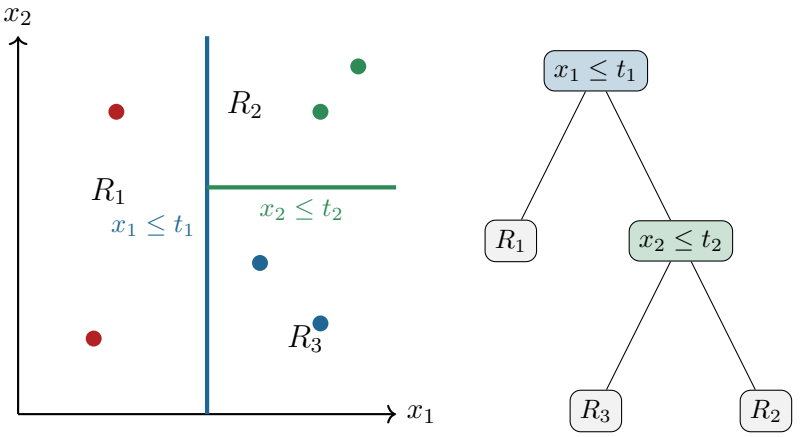
2.7.1 Geometric View: Axis-Aligned Splits

A decision tree divides the feature space into **rectangular regions**. Mathematically, it forms a **piecewise constant function**.

At any internal node, the split rule is:

$$\text{Is } x_j \leq t \ ?$$

$$R_{\text{left}} = \{x \mid x_j \leq t\}, \qquad R_{\text{right}} = \{x \mid x_j > t\}$$



Key Insight: Decision Trees always split **parallel to feature axes**. This makes them simple, interpretable, and easy to visualize.

2.7.2 Choosing the Best Split: Impurity Measures

At every internal node, a decision tree must decide:

"Which feature and threshold should I split on?"

The goal is to create child nodes that are **more homogeneous (pure)** than the parent node. A node is said to be **pure** if it contains samples from *only one class.*

To measure how mixed or pure a node is, decision trees use **impurity measures**. Lower impurity means better class separation.

Common Impurity Measures

A. Entropy (Information Theory)
Entropy originates from **information theory** and quantifies the amount of **uncertainty or disorder** in a dataset. In decision trees, it tells us how difficult it is to predict the class label at a node.

$$H(S) = -\sum_{k=1}^{K} p_k \log_2 p_k$$

- p_k represents the proportion of class k in node S
- $H(S) = 0$ indicates **complete certainty** (pure node)
- Higher entropy means higher class confusion

Common Impurity Measures

Example (Spam Detection):
Consider an email inbox classifier:

- If all emails in a folder are *Spam*, entropy is 0
- If emails are evenly split between *Spam* and *Not Spam*, entropy is maximum

This explains why decision trees prefer splits that separate spam and non-spam emails clearly.

B. Gini Impurity (CART Standard)
Gini impurity measures the likelihood that a randomly selected sample from a node would be **misclassified** if its label were assigned randomly based on the node's class distribution.

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

- $G(S) = 0$ corresponds to a completely pure node
- Larger values indicate greater overlap between classes

Example (Medical Diagnosis):
Suppose a node contains patients classified as *Diseased* or *Healthy*:

- If almost all patients are diseased, Gini impurity is very low
- If patients are equally split, the impurity is high

Decision trees aim to reduce this impurity to make **reliable medical decisions**.

How the Split is Chosen
At each node, the decision tree evaluates all possible questions of the form: *"Should we split the data using this feature at this threshold?"* For every candidate split, the tree:

1. Computes the impurity of the parent node
2. Computes the impurity of the left and right child nodes
3. Chooses the split that **minimizes the weighted impurity**

The decrease in impurity achieved after the split is called **Information Gain**. **Example (Loan Approval System):** Consider a bank deciding whether to approve loans:

- Parent node contains both *Approved* and *Rejected* loans
- The tree tests a split such as: *"Is income* $\leq$ *$50,000?"*
- If this split separates most approved and rejected cases, impurity drops

The split that produces the **largest reduction in uncertainty** (i.e., highest information gain) is selected.

Practical Insight for AI learners:

- Entropy and Gini usually lead to **similar splits**
- Gini impurity is computationally simpler and faster
- `scikit-learn` uses **Gini** by default

Numerical Comparison Example
Suppose a node contains 10 samples:

$$6 \text{ positive}, \quad 4 \text{ negative}$$

Entropy:

$$H = -\left(\frac{6}{10}\log_2\frac{6}{10} + \frac{4}{10}\log_2\frac{4}{10}\right) \approx 0.971$$

Gini Impurity:

$$G = 1 - \left(\frac{6}{10}\right)^2 - \left(\frac{4}{10}\right)^2 = 0.48$$

Now consider a better split producing two child nodes:

- Left node: 5 positive, 1 negative
- Right node: 1 positive, 3 negative

Weighted Entropy and **Weighted Gini** both decrease after the split, indicating improved class purity.

Key Observation:
Although the **numerical values differ**, both impurity measures **rank splits in the same order**. This is why they often select the same split in practice.

Rule of Thumb:

- Entropy → ID3, C4.5
- Gini → CART (used in sklearn)

Example: Information Gain

Overview

To understand how a decision tree selects the best feature for splitting, let us walk through a simple Example.

Assume we have a dataset of **14 training samples**:

9 **Yes** and 5 **No**

Here, **Yes** and **No** represent two possible class labels.

Step 1: Compute the Entropy of the Parent Node

The entropy of the parent node tells us how **mixed** the class labels are before making any split.
In our dataset, we have a total of 14 samples:

$$9 \text{ Yes} \quad \text{and} \quad 5 \text{ No}$$

Step 1.1: Compute Class Probabilities

$$p(\text{Yes}) = \frac{9}{14}, \qquad p(\text{No}) = \frac{5}{14}$$

Step 1.2: Apply the Entropy Formula The entropy formula for a binary classification problem is:

$$H(S) = -\left[p(\text{Yes}) \log_2 p(\text{Yes}) + p(\text{No}) \log_2 p(\text{No})\right]$$

Substituting the values:

$$H(S) = -\left[\frac{9}{14} \log_2 \left(\frac{9}{14}\right) + \frac{5}{14} \log_2 \left(\frac{5}{14}\right)\right]$$

Step 1.3: Numerical Calculation

$$\log_2 \left(\frac{9}{14}\right) \approx -0.64, \qquad \log_2 \left(\frac{5}{14}\right) \approx -1.49$$

$$H(S) = -\left[\frac{9}{14}(-0.64) + \frac{5}{14}(-1.49)\right]$$

$$H(S) = -\left[-0.411 - 0.532\right] = 0.943$$

Final Result:

$$\boxed{H(S) \approx 0.94}$$

Interpretation: Since the entropy value is close to 1, the dataset is **quite mixed**, meaning there is significant uncertainty before making any split. This value indicates that the dataset contains a **high level of uncertainty**.

Step 2: Try a Split Using the Feature "Wind"

Now we test whether the feature *Wind* helps reduce this uncertainty. The dataset is divided into two groups:

- **Wind = Weak:** 8 samples (6 Yes, 2 No)
- **Wind = Strong:** 6 samples (3 Yes, 3 No)

Step 3: Compute Entropy After the Split

- Entropy of *Wind* = *Weak* group: $H = 0.81$
- Entropy of *Wind* = *Strong* group: $H = 1.00$

A value of $H = 1.00$ means the classes are **perfectly mixed**.

Step 4: Compute Information Gain

Information Gain tells us how much uncertainty is removed after the split. It is calculated as:

$$IG(S, \text{Wind}) = H(S) - \sum_{v} \frac{|S_v|}{|S|} H(S_v)$$

Substituting the values:

$$IG = 0.94 - \left(\frac{8}{14} \times 0.81 + \frac{6}{14} \times 1.00 \right) = \boxed{0.048}$$

Final Interpretation: The Information Gain is very small. This means the feature *Wind* does **not significantly reduce uncertainty**, so it is **not a strong choice** for splitting at this stage.

Interpretation: Wind reduces uncertainty **slightly**, so it is not the best splitting feature.

2.7.3 Overfitting and Regularization

Decision Trees are powerful learning models because they can form very detailed and flexible decision boundaries. However, this same flexibility can

become a weakness, making decision trees **highly prone to overfitting**.

The Problem: Overfitting

Overfitting occurs when a tree grows so deep that it starts memorizing the training data, including random noise and outliers. As a result, the model performs very well on training data but **poorly on unseen test data**.

Numerical Illustration:
Suppose we train a decision tree on 100 samples:

- A very deep tree achieves **0% training error**
- The same tree produces **25% test error**

This large gap between training and test performance is a clear sign of overfitting.

To reduce overfitting, we apply **regularization techniques** that control the size and complexity of the tree.

1. Pre-Pruning (Early Stopping)

In pre-pruning, the tree growth is intentionally stopped *before* it becomes overly complex.

- `max_depth`: limits how many splits the tree can make
- `min_samples_leaf`: requires a minimum number of samples in each leaf

Example:

- Without constraints: tree grows to depth 12 with 60 leaf nodes
- With `max_depth = 4`: tree has only 10 leaf nodes

The smaller tree may slightly increase training error but usually **improves generalization**.

2. Post-Pruning (Pruning After Training)

In post-pruning, a large tree is first allowed to grow fully. Afterwards, weak branches that contribute little to prediction accuracy are removed.

A widely used post-pruning method is **Cost-Complexity Pruning**, which balances accuracy and simplicity.

Cost-Complexity Objective Function

The selection process is governed by the following formula:

$$R_\alpha(T) = \text{Training Error}(T) + \alpha \times \text{Number of Leaf Nodes}$$

Example:

Consider two candidate trees:

- **Tree** T_1: Training Error $= 8$, Leaf Nodes $= 30$
- **Tree** T_2: Training Error $= 10$, Leaf Nodes $= 10$

For $\alpha = 0.2$:

$$R_\alpha(T_1) = 8 + 0.2 \times 30 = 14$$

$$R_\alpha(T_2) = 10 + 0.2 \times 10 = 12$$

Since $R_\alpha(T_2) < R_\alpha(T_1)$, the simpler tree T_2 is preferred.

Interpretation

Although T_2 has slightly higher training error, it is simpler and less likely to overfit, making it a better choice for real-world data.

Key Takeaway

The goal of regularization is not to fit the training data perfectly, but to learn a model that is **accurate, stable, and generalizable**.

2.7.4 Strengths and Limitations

Property	Explanation	Assessment
Scale Invariance	Splits depend on feature order, not magnitude	Strong
Nonlinearity	Learns complex decision boundaries automatically	Strong
Stability	Small data changes may produce different trees	Weak
Greedy Nature	Locally optimal splits may not yield global optimum	Weak

☞ Practical Advice:

Single Decision Trees are mainly **learning tools**. Real-world systems rely on ensembles:

- **Random Forests** → Many trees, less variance
- **Gradient Boosting** → Sequential error correction

```python
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from scipy.stats import entropy

# ------------------------------------------------
# Example 1: Entropy and Gini Calculation
# ------------------------------------------------

# Class distribution: 9 Yes, 5 No
labels = np.array([1]*9 + [0]*5)

# Entropy
```

```
probabilities = np.bincount(labels) / len(labels)
entropy_value = entropy(probabilities, base=2)

# Gini Impurity
gini_value = 1 - np.sum(probabilities2)

print("Entropy:", round(entropy_value, 3))
print("Gini Impurity:", round(gini_value, 3))

# ------------------------------------------------
# Example 2: Information Gain Calculation
# ------------------------------------------------

# Parent entropy
parent_entropy = entropy_value

# Child nodes after split
left_child = np.array([1]*6 + [0]*2)
right_child = np.array([1]*3 + [0]*3)

left_prob = np.bincount(left_child) / len(left_child)
right_prob = np.bincount(right_child) / len(right_child)

left_entropy = entropy(left_prob, base=2)
right_entropy = entropy(right_prob, base=2)

weighted_entropy = (
    (len(left_child)/len(labels)) * left_entropy +
    (len(right_child)/len(labels)) * right_entropy
)

information_gain = parent_entropy - weighted_entropy

print("Information Gain:", round(information_gain, 3))

# ------------------------------------------------
# Example 3: Decision Tree with and without Pruning
# ------------------------------------------------

# Synthetic dataset
X = np.random.rand(100, 2)
y = (X[:, 0] + X[:, 1] > 1).astype(int)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# Overfitted Tree
tree_overfit = DecisionTreeClassifier(criterion="gini")
tree_overfit.fit(X_train, y_train)

# Pruned Tree
```

```
tree_pruned = DecisionTreeClassifier(
    criterion="gini",
    max_depth=4,
    min_samples_leaf=5
)
tree_pruned.fit(X_train, y_train)

# Accuracy Comparison
acc_overfit = accuracy_score(y_test, tree_overfit.predict(X_test))
acc_pruned = accuracy_score(y_test, tree_pruned.predict(X_test))

print("Overfitted Tree Accuracy:", round(acc_overfit, 3))
print("Pruned Tree Accuracy:", round(acc_pruned, 3))
```

Listing 2.2: Python Implementation of Decision Tree Concepts

2.8 Random Forests

Random Forests are ensemble learning models that improve the performance of decision trees by combining the predictions of many trees. To understand how Random Forests work intuitively and mathematically, we will follow a **single real-world Example** throughout this section.

Running Example (Used Throughout)

Suppose we want to predict whether a customer will **buy a product (Yes / No)** based on the following features:

- Age
- Monthly Income
- Number of Website Visits
- Time Spent on Website

We have a dataset of **100 customers**.

2.8.1 Why Not a Single Decision Tree?

If we train a **single decision tree** on this dataset, it may create very specific rules such as:

> *If Age $<$ 23 and Income $>$ 40k and Visits $>$ 7, then Buy $=$ Yes*

Such rules may perfectly classify the training data but fail on new customers. This is a classic case of **overfitting**.

Decision trees have:

- **Low bias** (they learn complex patterns)
- **High variance** (they change a lot with small data variations)

> **Key Solution:** Random Forests solve this problem by combining **many different trees**.

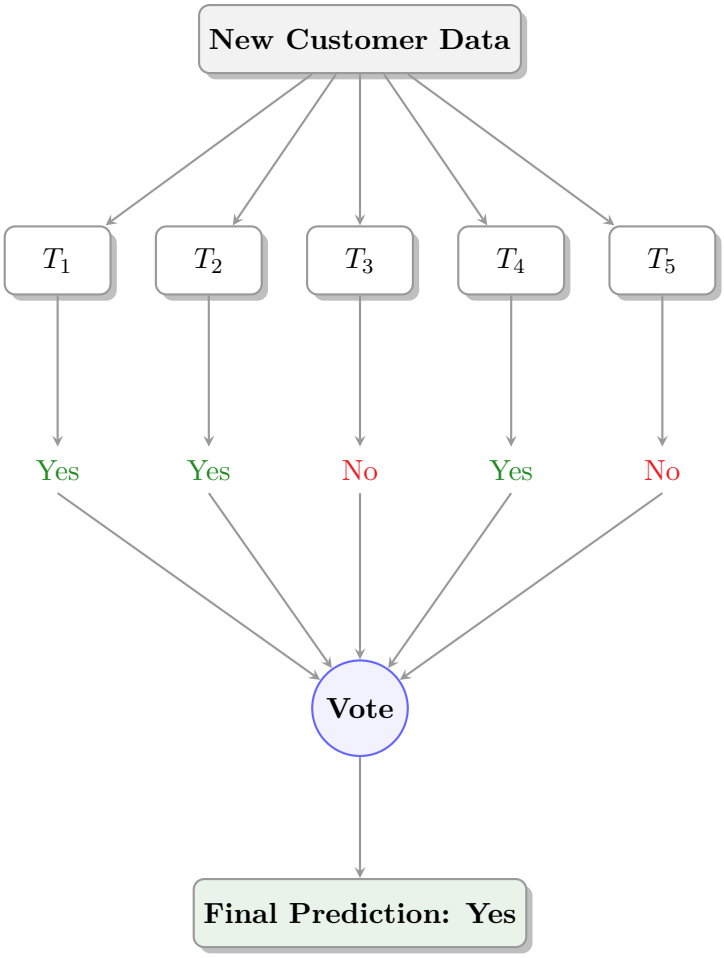
2.8.2 Ensemble Learning with a Example

Assume we train **5 different decision trees**. Each tree makes its own prediction for a new customer:

New Customer Profile

- Age = 30
- Income = 60k
- Visits = 5
- Time Spent = 8 minutes

Predictions from Individual Trees:



Tree	Prediction
T_1	Yes
T_2	Yes
T_3	No
T_4	Yes
T_5	No

Final Random Forest Prediction:

$$\hat{y} = \text{mode}(\text{Yes}, \text{Yes}, \text{No}, \text{Yes}, \text{No}) = \boxed{\text{Yes}}$$

This majority voting reduces the impact of incorrect individual trees.

Bootstrap Sampling (Bagging): A Numerical Explanation

Random Forests rely on a technique called **bootstrap sampling**, also known as **bagging (Bootstrap Aggregating)**, to create diversity among decision trees.

Assume the original training dataset contains **100 customer records**. For each decision tree in the forest:

- A new dataset of size **100** is generated
- Sampling is performed **with replacement**
- The same customer may appear multiple times
- Some customers may not be selected at all

Mathematically, the probability that a specific customer is *not* selected in a single draw is:

$$1 - \frac{1}{100}$$

After 100 draws, the probability that the customer is never selected becomes:

$$\left(1 - \frac{1}{100}\right)^{100} \approx 0.37$$

This means that, on average:

37% of customers are excluded from each tree's training set

Concrete Example:

- Tree T_1 may never see Customer #12
- Tree T_2 may never see Customer #47

These excluded samples are called **Out-of-Bag (OOB) samples**.

Why this matters: Each tree is trained on a slightly different dataset, ensuring variability in learned decision boundaries and reducing overfitting.

Random Feature Selection at Each Split

In addition to data randomness, Random Forests introduce randomness in **feature selection at every decision node**.

Assume each customer is described using:

$$d = 4 \quad \text{features}$$

For example:

{Age, Income, Visits, Time Spent}

At each split, the algorithm randomly selects:

$$m = \sqrt{d} = \sqrt{4} = 2 \quad \text{features}$$

Illustrative Split Decisions:

- **Tree** T_1: evaluates {Age, Income}
- **Tree** T_2: evaluates {Visits, Time Spent}
- **Tree** T_3: evaluates {Income, Visits}

Even when classifying the same customer, different trees:

- Examine different features
- Use different thresholds
- Produce independent decision paths

This deliberate randomness **reduces correlation among trees**, which is crucial for ensemble performance.

Out-of-Bag (OOB) Error Estimation

Out-of-Bag samples allow Random Forests to estimate prediction performance **without requiring a separate validation set**.

Suppose Customer #12:

- Is excluded from Tree T_1 during bootstrap sampling
- Can therefore be evaluated using T_1

This process is repeated:

- For all OOB customers

- Across all trees where they were excluded

Assume that, out of 100 customers:

$$85 \quad \text{are correctly predicted using OOB voting}$$

$$\textbf{OOB Accuracy} = \frac{85}{100} = \boxed{85\%}$$

This accuracy closely approximates real-world generalization performance.

Key Insight: OOB error provides an unbiased, efficient performance estimate at no additional computational cost.

Feature Importance: Numerical Interpretation

Random Forests quantify how much each feature contributes to decision-making by measuring the **total reduction in impurity** across all trees.

Assume the following normalized importance scores are obtained:

Feature	Importance Score
Time Spent	**0.40**
Visits	**0.30**
Income	**0.20**
Age	**0.10**

Interpretation:

- Time spent on the website dominates purchasing behavior
- Visit frequency plays a significant supporting role
- Age contributes minimally to prediction decisions

Such insights help organizations focus on the most influential customer attributes.

Advantages Demonstrated by the Example

From this customer prediction scenario, Random Forests offer:

- **Higher accuracy** through majority voting
- **Reduced variance** compared to single decision trees
- **Robustness to noisy data and outliers**
- **Interpretability** via feature importance analysis

Final Takeaway

By combining bootstrap sampling, random feature selection, and ensemble averaging, Random Forests convert unstable decision trees into a **powerful, reliable, and interpretable learning model**.

```
# ----------------------------------------------------
# Random Forest Example: Customer Purchase Prediction
# Features:
# 1. Age
# 2. Monthly Income
# 3. Number of Website Visits
# 4. Time Spent on Website (minutes)
# Target:
# Buy Product (1 = Yes, 0 = No)
# ----------------------------------------------------

import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# -----------------------------
# Step 1: Create Dataset
# -----------------------------
# Each row: [Age, Income, Visits, TimeSpent]
X = np.array([
    [22, 40000, 10, 15],
    [25, 42000, 8, 12],
    [30, 60000, 5, 8],
    [35, 65000, 4, 6],
    [45, 80000, 2, 4],
    [28, 52000, 7, 10],
    [40, 75000, 3, 5],
    [32, 58000, 6, 9],
    [27, 48000, 9, 14],
```

```
    [50, 90000, 1, 3]
])

# Labels: 1 = Buy, 0 = Not Buy
y = np.array([1, 1, 1, 0, 0, 1, 0, 1, 1, 0])

# ----------------------------
# Step 2: Train-Test Split
# ----------------------------
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# ----------------------------
# Step 3: Train Random Forest
# ----------------------------
rf_model = RandomForestClassifier(
    n_estimators=100,      # number of trees
    max_depth=4,           # tree depth control
    min_samples_leaf=2,    # regularization
    random_state=42
)

rf_model.fit(X_train, y_train)

# ----------------------------
# Step 4: Model Evaluation
# ----------------------------
y_pred = rf_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

print("Random Forest Accuracy:", round(accuracy, 2))

# ----------------------------
# Step 5: New Customer Prediction
# ----------------------------
# New customer:
# Age = 30, Income = 60000, Visits = 5, TimeSpent = 8
new_customer = np.array([[30, 60000, 5, 8]])
prediction = rf_model.predict(new_customer)

print("Prediction for New Customer:",
      "Buy" if prediction[0] == 1 else "Not Buy")

# ----------------------------
# Step 6: Feature Importance
# ----------------------------
features = ["Age", "Income", "Visits", "Time Spent"]
importances = rf_model.feature_importances_

for f, imp in zip(features, importances):
```

```
82     print(f"{f}: {round(imp, 3)}")
```

Listing 2.3: Random Forest Example: Customer Purchase Prediction

2.9 Real-World Applications of Supervised Learning

Practical Importance of Supervised Learning

Supervised learning allows intelligent systems to generate accurate predictions by learning patterns from labeled historical data.

Supervised learning is extensively used in real-world artificial intelligence applications where high-quality labeled data is available. By learning from examples provided by experts, these models can identify patterns and make decisions that closely approximate human judgment.

1. Medical Diagnosis (Disease Prediction)

Disease Prediction Using Patient Records

Supervised learning supports healthcare professionals by predicting diseases from clinical and demographic data.

In medical diagnosis, patient attributes such as laboratory test results, vital signs, and demographic information serve as input features, while the presence or absence of a disease is used as the target label. Once trained, the model can estimate disease risk for new patients.

Disease Risk Classification

Consider a supervised binary classification problem in which patient health indicators are used to predict the presence of a disease. Each patient record consists of two numerical features:

- Systolic blood pressure (mmHg)
- Cholesterol level (mg/dL)

The training dataset is defined as:

$$X = \begin{bmatrix} 120 & 85 \\ 140 & 95 \\ 160 & 100 \end{bmatrix}, \quad Y = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

where:

- $X \in \mathbb{R}^{3\times 2}$ represents patient feature vectors,
- $Y \in \{0, 1\}$ denotes the class labels,
- 0 corresponds to a healthy patient,
- 1 indicates the presence of disease.

Logistic regression learns a decision boundary of the form:

$$P(y = 1 \mid \mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$$

where $\sigma(\cdot)$ is the sigmoid function, $\mathbf{w}$ is the weight vector, and b is the bias term.

For a new patient with measurements:

$$\mathbf{x}_{\text{new}} = (150,\ 98),$$

the trained model estimates a high probability of disease and produces the prediction:

$$\hat{y} = 1$$

This indicates that the patient is classified as being at risk and may require further medical evaluation.

```
from sklearn.linear_model import LogisticRegression
import numpy as np

# Training data: [Blood Pressure, Cholesterol]
X = np.array([
    [120, 85],
    [140, 95],
    [160, 100]
])

# Labels: 0 = Healthy, 1 = Disease
y = np.array([0, 0, 1])

# Initialize logistic regression model
model = LogisticRegression(solver='liblinear')

# Train the model
model.fit(X, y)

# New patient data
new_patient = np.array([[150, 98]])

# Predict class label
```

```
prediction = model.predict(new_patient)

# Predict disease probability
probability = model.predict_proba(new_patient)

print("Predicted Class:", prediction)
print("Probability [Healthy, Disease]:", probability)
```

Listing 2.4: Medical Diagnosis using Logistic Regression

Interpretation of Results The output probability provides insight into the model's confidence. If the probability of disease exceeds a predefined threshold (commonly 0.5), the patient is classified as high-risk. Such predictive models assist clinicians by serving as decision-support tools, improving early detection and prioritization of medical resources.

2. Credit Scoring

Financial Risk Assessment and Fraud Detection

Supervised learning models are used by financial institutions to evaluate credit risk and identify fraudulent behavior.

Credit scoring systems classify loan applicants as either low-risk or high-risk based on historical financial data. Similarly, fraud detection systems flag unusual transaction patterns that deviate from normal behavior.

Credit Scoring and Loan Approval Consider a supervised classification problem in which a financial institution evaluates loan applications based on historical customer data. Each applicant is described using the following features:

- Annual income (in USD)
- Number of existing credit defaults

The training dataset is represented as:

$$X = \begin{bmatrix} 50000 & 2 \\ 30000 & 6 \\ 80000 & 0 \end{bmatrix}, \quad Y = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

where:

- $X \in \mathbb{R}^{3\times 2}$ denotes applicant feature vectors,

- $Y \in \{0, 1\}$ represents loan decisions,
- 1 indicates loan approval,
- 0 indicates loan rejection.

A decision tree classifier learns a sequence of hierarchical decision rules that partition the feature space into regions associated with different outcomes. These rules are typically of the form:

$$\text{Income} > \tau_1 \quad \text{and} \quad \text{Defaults} \leq \tau_2$$

For a new applicant with the attributes:

$$\mathbf{x}_{\text{new}} = (60000,\ 1),$$

the trained model evaluates the learned decision rules and produces the prediction:

$$\hat{y} = 1$$

This result suggests that the applicant satisfies the institution's approval criteria and is classified as low risk.

```python
from sklearn.tree import DecisionTreeClassifier
import numpy as np

# Training data: [Annual Income, Number of Defaults]
X = np.array([
    [50000, 2],
    [30000, 6],
    [80000, 0]
])

# Labels: 1 = Approved, 0 = Rejected
y = np.array([1, 0, 1])

# Initialize decision tree model
model = DecisionTreeClassifier(
    criterion='gini',
    max_depth=3,
    random_state=42
)

# Train the model
model.fit(X, y)

# New applicant data
```

```
new_applicant = np.array([[60000, 1]])

# Predict approval decision
prediction = model.predict(new_applicant)

# Predict class probabilities
probability = model.predict_proba(new_applicant)

print("Predicted Decision:", prediction)
print("Probability [Rejected, Approved]:", probability)
```

Listing 2.5: Credit Approval using Decision Tree

Interpretation of Results The decision tree evaluates the applicant by applying learned threshold-based rules derived from historical data. The predicted probability reflects the model's confidence in the approval decision. Such models are widely used in financial systems due to their interpretability, as decision paths can be visualized and audited for regulatory compliance.

3. Handwriting Recognition

Understanding Human Communication

Supervised learning transforms raw audio and image data into meaningful linguistic information.

Speech and handwriting recognition systems are trained using labeled audio samples or images. Each input is associated with a specific word, character, or digit, enabling accurate recognition.

Pixel-Based Representation of Handwritten Digits

Handwritten digit recognition is a supervised learning task in which an image of a digit is transformed into a numerical representation and classified into one of the classes $\{0, 1, \ldots, 9\}$. A widely used and intuitive approach represents each digit image as a grid of pixels, where every pixel contributes a binary or grayscale value to the feature space.

Pixel Grid and Feature Encoding Figure 2.13 illustrates a simplified 5×5 pixel grid representation of the handwritten digit "3". Each cell corresponds to a pixel location, and a filled cell indicates the presence of ink at that position.

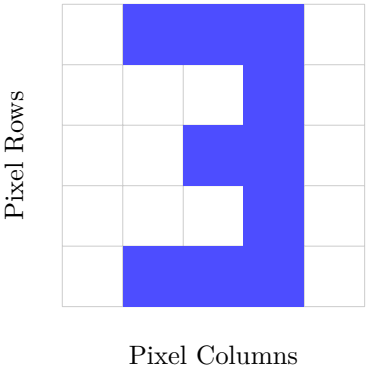


Figure 2.13: Pixel grid representation of a handwritten digit. Filled cells denote active pixels.

Each pixel at position (i, j) is encoded as a binary feature:

$$x_{ij} = \begin{cases} 1, & \text{if ink is present at pixel } (i, j), \\ 0, & \text{otherwise.} \end{cases}$$

For the digit "3" shown in Figure 2.13, the resulting feature matrix is:

$$\mathbf{X} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

This matrix is flattened row-wise to form a fixed-length feature vector:

$$\mathbf{x} = [0, 1, 1, 1, 0,\ 0, 0, 0, 1, 0,\ 0, 0, 1, 1, 0,\ 0, 0, 0, 1, 0,\ 0, 1, 1, 1, 0]^\top \in \mathbb{R}^{25}$$

Thus, each handwritten digit image is represented as a numerical vector suitable for machine learning models.

Supervised Dataset Formulation A supervised learning dataset is constructed as a collection of labeled examples:

$$\mathcal{D} = \left\{ (\mathbf{x}^{(i)}, y^{(i)}) \right\}_{i=1}^{N}$$

where $\mathbf{x}^{(i)} \in \mathbb{R}^{25}$ denotes the pixel-based feature vector and $y^{(i)} \in \{0, 1, \ldots, 9\}$ represents the corresponding digit label.

Learning and Classification Model The objective of learning is to approximate a function

$$f : \mathbb{R}^{25} \to \{0, 1, \ldots, 9\}$$

that maps pixel features to digit classes.

In probabilistic classifiers such as logistic or softmax regression, a score is computed for each class k. For example, the class probability can be expressed as:

$$P(y = k \mid \mathbf{x}) = \frac{\exp(\mathbf{w}_k^\top \mathbf{x} + b_k)}{\sum_{j=0}^{9} \exp(\mathbf{w}_j^\top \mathbf{x} + b_j)}$$

where $\mathbf{w}_k$ and b_k are learned parameters for class k.

The predicted digit is obtained using the maximum probability rule:

$$\hat{y} = \arg\max_k P(y = k \mid \mathbf{x})$$

Inference and Example: Digit "3" During inference, a new handwritten digit image undergoes the same processing steps: pixel discretization, binary feature extraction, vector formation, and classification by the trained model.

For the digit "3" illustrated earlier, the extracted feature vector closely matches the learned spatial patterns associated with class "3". The combination of horizontal strokes at the top and bottom and a vertical stroke on the right side enables the classifier to distinguish it from visually similar digits such as "5" or "8". Consequently, the model assigns the highest probability to class "3", yielding:

$$\hat{y} = 3$$

This example demonstrates how a simple pixel-based representation allows machine learning models to recognize handwritten digits by learning discriminative spatial structures from data.

4. Automatic Speech Recognition

Decoding the Acoustic Signal

Automatic Speech Recognition (ASR) is the supervised learning task of converting a continuous, unstructured audio stream into a structured sequence of text characters or words.

For a computer, an audio file is simply a long list of numbers representing air pressure changes. To recognize speech, the system must learn to ignore background noise and speaker accents to focus on the fundamental linguistic patterns.

The Speech Recognition Pipeline

The conversion process is structured as a pipeline that reduces data complexity at each step. By the time the data reaches the machine learning model, the "raw noise" has been transformed into a "mathematical thumbprint" of the spoken words.

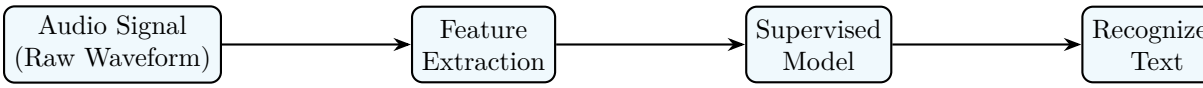


Figure 2.14: The modular architecture of a modern speech recognition system.

From Waveforms to Features (Frequency Analysis) If we look at a raw audio signal (the *Time Domain*), it is difficult to see the difference between the sound "S" and the sound "O". However, every sound has a unique "signature" based on its pitch and harmonics.

To extract these signatures, we use Feature Extraction. Instead of looking at the signal over time, we look at it over **frequencies** (the *Frequency Domain*). This is similar to how a prism breaks white light into a rainbow; we break the audio into its component pitches.

- **Framing:** The audio is cut into tiny slices (frames) of roughly 25ms.
- **Mel-Scaling:** Humans are better at hearing differences in low pitches than high pitches. We adjust the data to match this human bias using a "Mel Scale."
- **Result:** Each slice of audio becomes a **Feature Vector x**, which acts like a column in a digital image of the sound.

From Vibrations to Vectors: A Example The most critical step in the pipeline is the conversion of raw audio into a **Feature Vector x**. While a raw audio file is a sequence of thousands of pressure values per second, the AI model requires a more "meaningful" summary of the sound's character.

To achieve this, the system slices the audio into overlapping **frames** (typically 25ms long). Each slice is transformed into a vector that represents the distribution of energy across different pitches.

Step 1: Frequency Binning

Imagine we divide the audible spectrum into five "bins" ranging from Low Pitch (Bass) to High Pitch (Treble). For a single 25ms slice where a speaker says the vowel sound “aa”, the extraction algorithm measures the intensity in each bin:

- **Bin 1 (Bass):** 0.10 (Low energy)
- **Bin 2 (Low-Mid):** 0.85 (Strong resonance)
- **Bin 3 (Mid):** 0.70 (Strong resonance)
- **Bin 4 (High-Mid):** 0.20 (Low energy)
- **Bin 5 (Treble):** 0.05 (Negligible energy)

Step 2: Vector Formulation
These values are stacked to form the feature vector for that specific moment in time:

$$\mathbf{x}^{(t)} = \begin{bmatrix} 0.10 \\ 0.85 \\ 0.70 \\ 0.20 \\ 0.05 \end{bmatrix} \in \mathbb{R}^5$$

Step 3: The "Sound Image" (Spectrogram)
As the speaker continues, a new vector is generated every 10ms. When these vectors are placed side-by-side, they form a **Spectrogram**=a digital image of the sound where the y-axis is frequency, the x-axis is time, and the pixel intensity is the value in the vector.

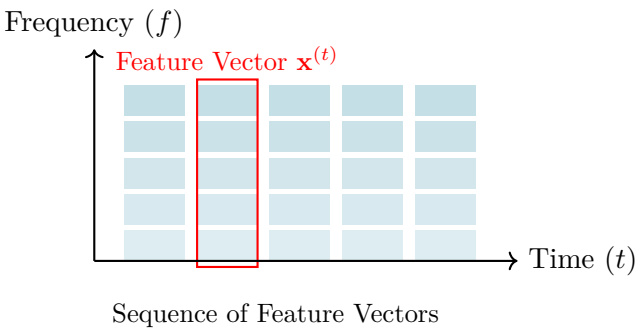


Figure 2.15: Conceptual spectrogram representation. Each vertical column corresponds to a feature vector $\mathbf{x}^{(t)}$ extracted from a short-time speech frame (e.g., 25 ms).

1. **Dimensionality Reduction:** We compress 400 raw samples (at 16kHz) into a single 5-dimensional vector, making the data easier for a Neural Network to process.
2. **Pattern Recognition:** The model learns that a "high-energy middle" (like our vector above) signifies a vowel, while high energy in the top bins signifies a "hissing" sound like “s” or “f”.

The Supervised Learning Task Once the audio is converted into a sequence of vectors $\mathbf{X} = [\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(T)}]$, the supervised model performs classification. It calculates the probability of a word w based on the observed patterns:

$$\hat{w} = \arg\max_{w} P(w \mid \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(T)})$$

By comparing the input sequence to thousands of labeled hours of human speech, the model identifies which word most likely produced that specific "visual" fingerprint in the spectrogram.

Data Representation and Dimensions As the data moves through the pipeline, it is compressed from millions of raw samples into a few highly informative coefficients, as detailed in Table 2.3.

Table 2.3: Data Evolution in the ASR Pipeline

Representation	**Structure**	**Analogy**
Raw Audio	1D Array	Similar to raw, unorganized sensor data.
Spectrogram	2D Matrix	Like a "heat map" where brightness represents the volume of a certain pitch.
Feature Vector	$\mathbb{R}^d$ Vector	A numerical "summary" of a 25ms slice of speech.
Output Text	String	The final classification label (sequence of words).

Supervised Learning: The Acoustic Model In the supervised learning phase, we provide the model with a dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}$, where $\mathbf{x}$ is the sequence of audio features and y is the correct transcript.

The model (often a Deep Neural Network) learns a mapping function $f(\mathbf{X}) \rightarrow \mathbf{y}$. Because speech happens over time, the model must consider the **context**. For example, if the model hears the sound "/b/", it is more likely that the next sound is a vowel (like "/a/") than another consonant (like "/z/").

We represent the probability of a word sequence $\mathbf{w}$ given the audio features $\mathbf{X}$ using the following logic:

$$P(\text{word} \mid \text{audio}) \propto P(\text{audio} \mid \text{word}) \times P(\text{word sequence context})$$

Inference and Transcription When a user speaks into a microphone, the system:

1. Converts the raw vibrations into a sequence of feature vectors.
2. Feeds these vectors into the trained Neural Network.
3. The model identifies the most likely "phonemes" (units of sound).
4. A **Language Model** joins these sounds into logical words.

For example, if the audio features match both "write" and "right," the language model looks at the surrounding words. If the previous words were "Please . . . this letter," the model assigns a higher probability to "write."

Final Output:

$$\hat{y} = \text{"Please write this letter"}$$

This combination of signal processing and probabilistic modeling allows AI to transform complex physical waves into meaningful digital text.

4. Face Recognition Systems

Facial Identity Recognition

Face recognition systems identify or verify individuals using learned facial features.

These systems are trained on labeled facial images and later classify new images based on similarity to known identities.

Example

Similarity-Based Classification of Visual Objects In many computer vision applications, such as face or object recognition, raw images are first transformed into compact numerical representations known as feature vectors or embeddings. These embeddings capture the most discriminative visual characteristics of an object while reducing dimensionality.

Classification is then performed by comparing the similarity between the feature representation of a new object and those stored in a labeled database.

Example (Object Recognition via Similarity Scores) Consider an object recognition system in which each image is represented by a similarity score obtained after feature extraction and normalization. The labeled training dataset is defined as:

$$X = \begin{bmatrix} 0.85 \\ 0.40 \\ 0.90 \end{bmatrix}, \quad Y = \begin{bmatrix} \text{Ali} \\ \text{Ahmed} \\ \text{Sara} \end{bmatrix}$$

where:

- X contains similarity scores of known objects or identities,
- Y represents their corresponding class labels,
- larger values indicate higher visual similarity.

For a new object image with similarity score:

$$x_{\text{new}} = 0.88,$$

the classifier computes the distance between the new input and each stored reference.

$$|0.88 - 0.85| = 0.03, \quad |0.88 - 0.40| = 0.48, \quad |0.88 - 0.90| = 0.02.$$

Based on the nearest neighbor decision rule, the model assigns the label corresponding to the closest similarity score:

$$\hat{y} = \text{Ali}$$

Decision Rule Using a nearest neighbor classifier, the predicted class is given by:

$$\hat{y} = \arg\min_i \left| x_{\text{new}} - x^{(i)} \right|$$

This simple yet effective rule forms the basis of many real-world recognition systems.

Object Image Representation Figure **??** illustrates a simplified human (male) icon representing an object or individual that may be processed by a recognition system. In practice, such visual inputs are converted into numerical feature vectors before being analyzed and classified by a supervised learning model.

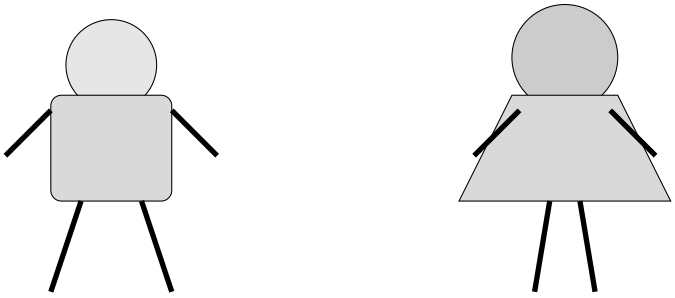


Figure 2.16: Simplified human icons representing male and female subjects placed side by side for object recognition and classification tasks.

Learning and Classification Process The complete learning and classification pipeline for object recognition is summarized as follows:

1. Capture the object image using a camera or sensor.
2. Extract discriminative features from the image.
3. Convert features into a numerical similarity representation.
4. Compare the new representation with stored labeled samples.
5. Assign the class label of the most similar reference object.

This similarity-based approach is widely used in face recognition, object identification, and biometric systems due to its simplicity, interpretability, and effectiveness.

```
1
2  import numpy as np
3  from sklearn.neighbors import KNeighborsClassifier
4  from sklearn.metrics.pairwise import cosine_similarity
5
6  # ------------------------------------------------
7  # Step 1: Training Data (Feature Representations)
8  # ------------------------------------------------
9  # Each value represents a similarity score or a
10 # reduced embedding obtained from an image encoder
11
12 X_train = np.array([
13     [0.85],   # Object / Face: Ali
14     [0.40],   # Object / Face: Ahmed
```

```
    [0.90]    # Object / Face: Sara
])

y_train = np.array([
    "Ali",
    "Ahmed",
    "Sara"
])

# ------------------------------------------------
# Step 2: Train a Nearest Neighbor Classifier
# ------------------------------------------------
model = KNeighborsClassifier(
    n_neighbors=1,
    metric='euclidean'
)

model.fit(X_train, y_train)

# ------------------------------------------------
# Step 3: New Object / Face Input
# ------------------------------------------------
X_test = np.array([[0.88]])

# ------------------------------------------------
# Step 4: Classification
# ------------------------------------------------
predicted_label = model.predict(X_test)

# Retrieve nearest neighbor distance and index
distances, indices = model.kneighbors(X_test)

# ------------------------------------------------
# Step 5: Output Results
# ------------------------------------------------
print("Predicted Identity:", predicted_label[0])
print("Nearest Reference Score:", X_train[indices[0][0]][0])
print("Distance to Nearest Neighbor:", distances[0][0])

# ------------------------------------------------
# Optional: Cosine Similarity (Common in Recognition)
# ------------------------------------------------
similarities = cosine_similarity(X_test, X_train)
best_match_index = np.argmax(similarities)

print("Cosine Similarity Scores:", similarities)
```

```
print("Best Match (Cosine):", y_train[best_match_index])
```

Listing 2.6: Similarity-Based Object Recognition using Nearest Neighbor

5. Autonomous Perception Modules

Perception in Autonomous Vehicles

Supervised learning enables autonomous systems to perceive and interpret their environment.

Autonomous vehicles rely on labeled sensor data from cameras and LiDAR systems to classify objects such as pedestrians, vehicles, and traffic signs. **Example:**

$$X = [1.2,\ 2.8,\ 0.6], \quad Y = [\text{Pedestrian},\ \text{Vehicle},\ \text{Traffic Sign}]$$

For a sensor reading of 1.1:

$$\hat{y} = \text{Pedestrian}$$

```
from sklearn.neighbors import KNeighborsClassifier

X = [[1.2],[2.8],[0.6]]
y = ["Pedestrian","Vehicle","Traffic Sign"]

model = KNeighborsClassifier(n_neighbors=1)
model.fit(X, y)

print("Detected Object:", model.predict([[1.1]]))
```

Listing 2.7: Autonomous Object Classification

Final Insight

Supervised learning effectively bridges theoretical machine learning concepts and real-world applications by enabling systems to learn from labeled data and make intelligent decisions across multiple domains.

Summary

Supervised Learning Summary:

- Uses labeled datasets
- Learns a mapping from inputs to outputs
- Includes regression and classification
- Requires evaluation and regularization
- Forms the backbone of modern AI systems

Chapter 3

Unsupervised Learning Algorithms

Learning from Unlabeled Data

Unsupervised learning discovers hidden structures and patterns in data without relying on labeled outputs.

Unsupervised Learning is a fundamental machine learning method in which models learn directly from data that does not contain explicit target labels. Unlike supervised learning, the algorithm is not guided by correct answers. Instead, it autonomously identifies meaningful structures, similarities, relationships, or latent representations within the dataset.

This learning method closely resembles human exploratory learning, where patterns are inferred through observation rather than instruction. Humans naturally group objects, detect anomalies, and recognize similarities without being explicitly told how to do so.

Unsupervised learning plays a critical role in data exploration, knowledge discovery, dimensionality reduction, anomaly detection, and representation learning, particularly when labeled data is scarce, expensive, or unavailable.

Formal Mathematical Definition

Let the unlabeled dataset be defined as:

$$\mathcal{D} = \{x_1, x_2, \ldots, x_n\}$$

where:

- $x_i \in \mathbb{R}^d$ represents a d-dimensional feature vector,
- no corresponding target labels are available.

The objective of unsupervised learning is to uncover an underlying structure or representation:

$$\mathcal{S} = g(\mathcal{D})$$

where $g(\cdot)$ denotes a learning process that may perform clustering, density estimation, or feature transformation.

Key Idea

The model learns from the intrinsic properties of the data itself, not from external supervision.

Major Categories of Unsupervised Learning

Unsupervised learning methods are commonly grouped into the following categories:

1. Clustering Clustering algorithms group similar data points into clusters based on a similarity or distance measure.

2. Dimensionality Reduction These techniques reduce the number of features while preserving important structure or variance in the data.

3. Density Estimation Models attempt to estimate the underlying probability distribution of the data.

4. Anomaly Detection Identifies rare or unusual data points that deviate significantly from normal patterns.

Unsupervised Learning Categories

Category	Goal	Example Algorithm
Clustering	Group similar samples	K-Means
Dimensionality Reduction	Reduce feature space	PCA
Density Estimation	Model data distribution	GMM
Anomaly Detection	Detect outliers	Isolation Forest

3.1 Clustering: Conceptual Overview

Clustering is a fundamental task in unsupervised learning in which the objective is to discover hidden structures or patterns in unlabeled data. In

contrast to supervised learning, no ground-truth labels are available. Instead, clustering algorithms analyze the intrinsic properties of the data and automatically organize samples based on similarity.

Formally, clustering partitions a dataset into multiple groups, known as *clusters*, such that:

- Samples within the same cluster exhibit high similarity.
- Samples belonging to different clusters are significantly dissimilar.

Intuition

In many real-world datasets, observations that are close to each other in the feature space tend to share common characteristics. Clustering algorithms aim to uncover these natural groupings without any prior labeling information.

To better understand this concept, Figure 3.1 provides a visual illustration of clustering in a two-dimensional feature space. Each point represents a data sample, and clusters emerge as dense regions of similar observations.

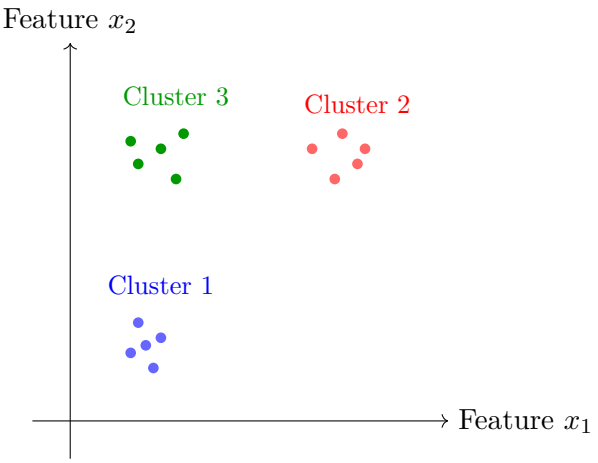


Figure 3.1: Conceptual illustration of clustering in a two-dimensional feature space. Points within the same cluster are close to each other, while points from different clusters are well separated.

The notion of *similarity* or *dissimilarity* between data points is central to clustering. This similarity is typically quantified using a distance metric. For continuous numerical features, the most commonly used metric is the Euclidean distance:

$$d(x_i, x_j) = \sqrt{\sum_{k=1}^{d} (x_{ik} - x_{jk})^2}$$

where:

- $x_i, x_j \in \mathbb{R}^d$ are two data points,
- d denotes the number of features,
- x_{ik} represents the k-th feature of data point x_i.

A smaller distance value indicates a higher degree of similarity between samples, whereas larger distances suggest dissimilar behavior. By leveraging such distance measures, clustering algorithms can effectively group unlabeled data into meaningful structures.

Example: K-Means Clustering

K-Means is one of the most widely used clustering algorithms in unsupervised learning due to its conceptual simplicity and computational efficiency. The primary objective of K-Means is to partition a dataset into K distinct clusters such that data points within the same cluster are as similar as possible, while points in different clusters are well separated.

Customer Segmentation Example

A practical illustration of clustering in the absence of labeled data, commonly employed in marketing analytics and customer behavior analysis.

Step 1: Dataset To illustrate clustering, consider a small dataset representing the purchasing behavior of customers in a store. We describe each customer using two numerical features:

- Annual spending on electronics (in thousands of USD)
- Annual spending on groceries (in thousands of USD)

Each customer can therefore be represented as a point in a two-dimensional feature space, where the horizontal axis corresponds to electronics spending and the vertical axis to groceries spending. For example, the dataset can be expressed as:

$$X = \begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 8 & 7 \\ 9 & 8 \end{bmatrix}$$

Here, each row corresponds to a customer, and each column corresponds to a feature. When plotted, these data points naturally form two distinct groups: one cluster representing customers with low spending on both categories, and another cluster representing customers with high spending. This clear separation provides an intuitive basis for applying clustering algorithms.

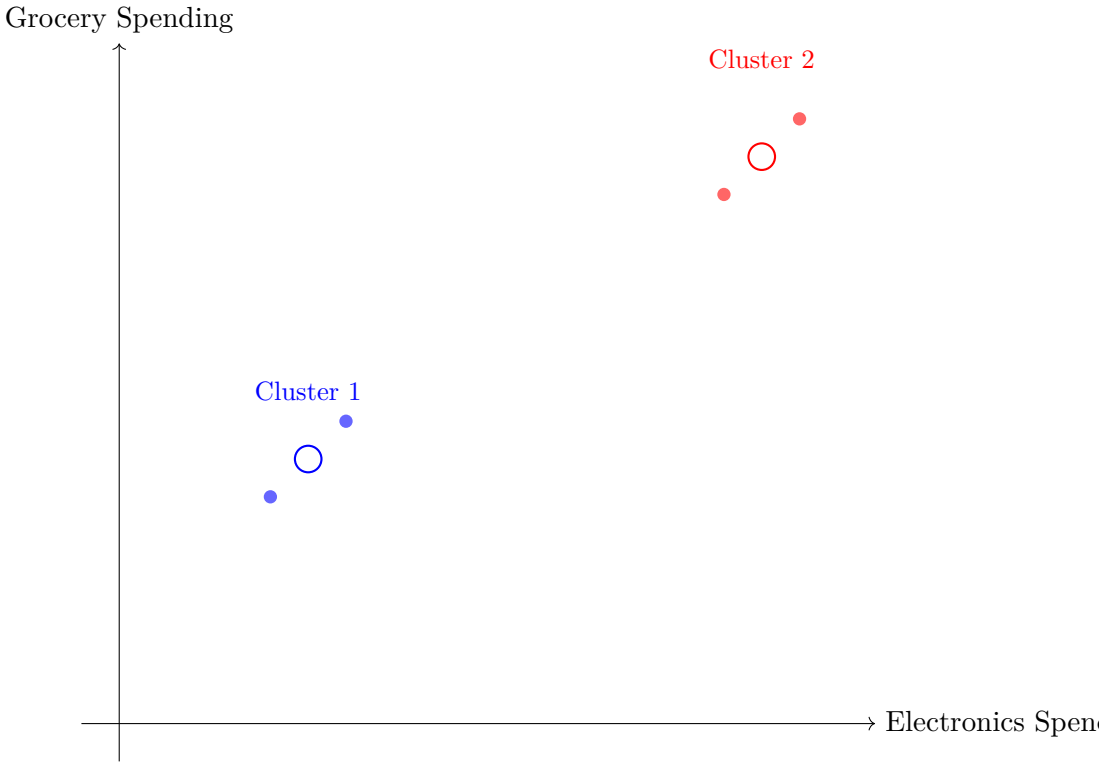


Figure 3.2: Illustration of K-Means clustering in a two-dimensional customer feature space. Circles denote cluster centroids, while colored points represent customer data samples.

Step 2: Choosing the Number of Clusters Before applying K-Means, the number of clusters K must be specified. In this example, based on visual inspection and domain intuition, we select:

$$K = 2$$

In real-world applications, methods such as the elbow method, silhouette score, or gap statistic are commonly used to determine an appropriate value of K.

Step 3: K-Means Objective Function The objective of K-Means is to minimize the *within-cluster sum of squares (WCSS)*, which measures the compactness of clusters:

$$\min \sum_{k=1}^{K} \sum_{x_i \in C_k} \|x_i - \mu_k\|^2$$

where:

- C_k represents the set of data points assigned to cluster k,
- μ_k denotes the centroid of cluster C_k,
- $\|\cdot\|$ is the Euclidean norm.

Each centroid is computed as the arithmetic mean of the data points in its cluster:

$$\mu_k = \frac{1}{|C_k|} \sum_{x_i \in C_k} x_i$$

Step 4: Iterative Optimization Process K-Means operates through an iterative refinement process involving two alternating steps:

1. **Assignment step**: Each data point is assigned to the nearest centroid based on Euclidean distance.
2. **Update step**: Centroids are recomputed as the mean of all data points assigned to each cluster.

These steps are repeated until cluster assignments stabilize or a predefined stopping criterion is met.

Step 5: Resulting Clusters After convergence, the algorithm produces the following clusters:

$$C_1 = \{(2,3),(3,4)\}, \quad C_2 = \{(8,7),(9,8)\}$$

Interpretation

The K-Means algorithm successfully discovers two distinct customer segments=low spenders and high spenders=purely based on feature similarity, without requiring any labeled training data.

Python Implementation: K-Means Clustering

The following Python code demonstrates how K-Means clustering can be implemented using the `scikit-learn` library.

```
from sklearn.cluster import KMeans
import numpy as np

# Customer data
X = np.array([
    [2, 3],
    [3, 4],
    [8, 7],
    [9, 8]
])

# Initialize K-Means model
model = KMeans(n_clusters=2, random_state=42)

# Fit model to data
model.fit(X)

# Extract cluster assignments
labels = model.labels_

# Display results
print("Cluster Labels:", labels)
print("Centroids:", model.cluster_centers_)
```

Listing 3.1: Customer Segmentation using K-Means

The output labels indicate the cluster membership of each customer, while the centroids represent the average spending behavior of each identified segment.

Key Insight

K-Means clustering transforms raw, unlabeled numerical data into meaningful groups, enabling data-driven decision-making in domains such as marketing, finance, and social sciences.

3.2 Dimensionality Reduction: Principal Component Analysis (PCA)

Dimensionality reduction is a key technique in unsupervised learning used to simplify complex datasets while retaining the essential information. Principal Component Analysis (PCA) is one of the most widely used methods for this purpose. It identifies directions (principal components) along which the data varies the most, and projects the data onto these directions to obtain a lower-dimensional representation.

Reducing Complexity

PCA transforms high-dimensional data into a lower-dimensional space while preserving maximum variance, enabling simpler visualization and more efficient learning.

Mathematical Formulation Given a centered data matrix $X \in \mathbb{R}^{n \times d}$ (with zero mean for each feature), the first principal component $\mathbf{w}_1$ is obtained by solving the optimization problem:

$$\max_{\mathbf{w}_1} \ \mathbf{w}_1^\top \Sigma \mathbf{w}_1 \quad \text{subject to } \|\mathbf{w}_1\| = 1$$

where:

- $\Sigma = \frac{1}{n} X^\top X$ is the covariance matrix of X,
- $\mathbf{w}_1$ is the unit vector that defines the direction of maximum variance,
- The constraint $\|\mathbf{w}_1\| = 1$ ensures the solution is a normalized direction vector.

Subsequent principal components $\mathbf{w}_2, \mathbf{w}_3, \ldots$ are computed similarly, with the additional requirement that they are orthogonal to all previously computed components. This orthogonality ensures that each principal component captures unique variance in the data.

Note: Understanding Eigenvectors and Eigenvalues

Eigenvectors and eigenvalues are key to PCA. An **eigenvector** shows a direction in which the data is stretched or compressed without changing its orientation, and the corresponding **eigenvalue** tells us how much the data is stretched along that direction.

Example:

Consider a 2×2 covariance matrix:

$$\Sigma = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

To find eigenvalues λ, solve the characteristic equation:

$$\det(\Sigma - \lambda I) = 0$$

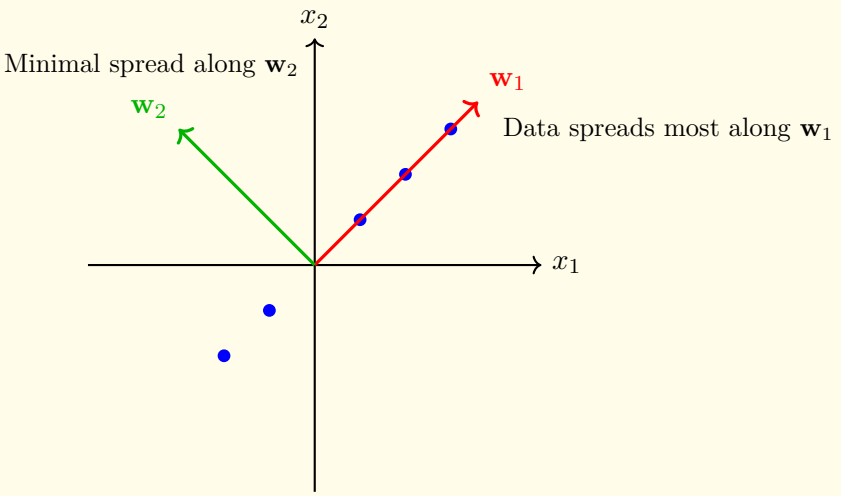
$$\begin{vmatrix} 1-\lambda & 1 \\ 1 & 1-\lambda \end{vmatrix} = (1-\lambda)^2 - 1 = 0$$

$$\Rightarrow \lambda^2 - 2\lambda = 0 \quad \Rightarrow \lambda_1 = 2, \quad \lambda_2 = 0$$

Eigenvectors $\mathbf{w}$ satisfy $(\Sigma - \lambda I)\mathbf{w} = 0$:

$$\text{For } \lambda_1 = 2: \quad \mathbf{w}_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \text{For } \lambda_2 = 0: \quad \mathbf{w}_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

This example illustrates that the principal component $\mathbf{w}_1$ points along the direction of maximal data spread, while $\mathbf{w}_2$ points along an orthogonal direction capturing minimal variance.



Step 1: Data Preparation Consider a small dataset:

$$X = \begin{bmatrix} 2 & 1 \\ 3 & 2 \\ 4 & 3 \end{bmatrix}$$

First, center the data by subtracting the mean of each feature:

$$\bar{X} = \begin{bmatrix} 2-3 & 1-2 \\ 3-3 & 2-2 \\ 4-3 & 3-2 \end{bmatrix} = \begin{bmatrix} -1 & -1 \\ 0 & 0 \\ 1 & 1 \end{bmatrix}$$

Intuition

Centering ensures that the PCA captures directions of variance relative to the data mean.

Step 2: Covariance Matrix Compute the covariance matrix Σ:

$$\Sigma = \frac{1}{n}\bar{X}^\top \bar{X} = \frac{1}{3}\begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 0.6667 & 0.6667 \\ 0.6667 & 0.6667 \end{bmatrix}$$

Note

The covariance matrix measures how much each feature varies with respect to others. Large off-diagonal values indicate strong correlation.

Step 3: Eigen Decomposition In PCA, the principal components are the directions of maximum variance in the data. Mathematically, these are the **eigenvectors** of the covariance matrix Σ:

$$\Sigma \mathbf{w} = \lambda \mathbf{w}$$

where:

- $\mathbf{w}$ is an eigenvector (principal component),
- λ is the corresponding eigenvalue representing variance along $\mathbf{w}$.

For the centered dataset:

$$\bar{X} = \begin{bmatrix} -1 & -1 \\ 0 & 0 \\ 1 & 1 \end{bmatrix}$$

the covariance matrix is

$$\Sigma = \begin{bmatrix} 0.6667 & 0.6667 \\ 0.6667 & 0.6667 \end{bmatrix}$$

Step 3a: Solving the Eigenvalue Problem We have the covariance matrix:

$$\Sigma = \begin{bmatrix} 0.6667 & 0.6667 \\ 0.6667 & 0.6667 \end{bmatrix}$$

To find eigenvalues λ, we solve the characteristic equation:

$$\det(\Sigma - \lambda I) = 0$$

$$\begin{vmatrix} 0.6667 - \lambda & 0.6667 \\ 0.6667 & 0.6667 - \lambda \end{vmatrix} = 0$$

Compute the determinant:

$$(0.6667 - \lambda)(0.6667 - \lambda) - (0.6667)(0.6667) = 0$$

$$(0.6667 - \lambda)^2 - 0.4444 = 0$$

$$0.4444 - 1.3334\lambda + \lambda^2 - 0.4444 = 0$$

$$\lambda^2 - 1.3334\lambda = 0$$

$$\lambda(\lambda - 1.3334) = 0$$

$$\Rightarrow \lambda_1 = 1.3333, \quad \lambda_2 = 0$$

Step 3b: Finding Eigenvectors Eigenvectors satisfy:

$$(\Sigma - \lambda I)\mathbf{w} = 0$$

For $\lambda_1 = 1.3333$:

$$\begin{bmatrix} 0.6667 - 1.3333 & 0.6667 \\ 0.6667 & 0.6667 - 1.3333 \end{bmatrix} \mathbf{w}_1 = \begin{bmatrix} -0.6666 & 0.6667 \\ 0.6667 & -0.6666 \end{bmatrix} \mathbf{w}_1 = 0$$

Solving this system, we get:

$$\mathbf{w}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad (\text{normalized: } \mathbf{w}_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix})$$

For $\lambda_2 = 0$:

$$\begin{bmatrix} 0.6667 - 0 & 0.6667 \\ 0.6667 & 0.6667 - 0 \end{bmatrix} \mathbf{w}_2 = \begin{bmatrix} 0.6667 & 0.6667 \\ 0.6667 & 0.6667 \end{bmatrix} \mathbf{w}_2 = 0$$

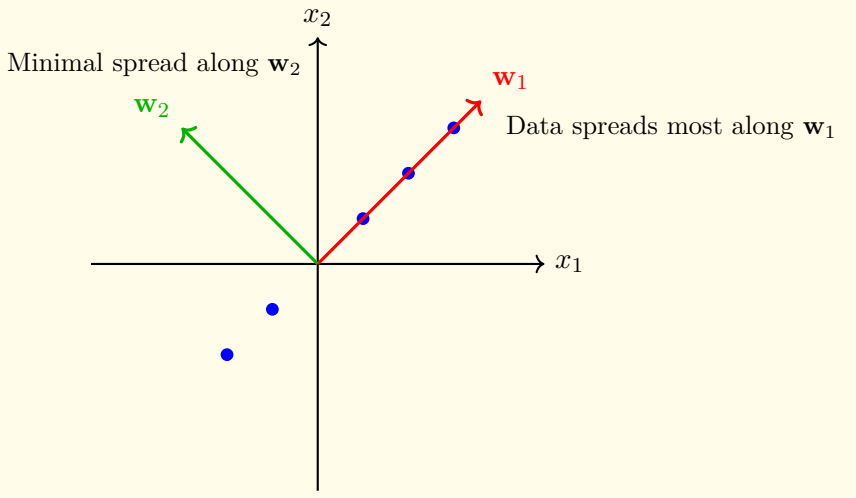
Solving this system, we get:

$$\mathbf{w}_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \quad (\text{normalized: } \mathbf{w}_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} -1 \\ 1 \end{bmatrix})$$

Thus, the principal components of the dataset are:

$$\lambda_1 = 1.3333, \ \mathbf{w}_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \lambda_2 = 0, \ \mathbf{w}_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

Visualization: Eigenvectors and Data Spread



Step 4: Projection onto Principal Components Once the eigenvectors are computed, the data can be projected onto the principal components to obtain a lower-dimensional representation:

$$Z = \bar{X}\mathbf{W}$$

where Z is the transformed dataset and $\mathbf{W} = [\mathbf{w}_1, \mathbf{w}_2]$ contains eigenvectors as columns. In this example, projecting along $\mathbf{w}_1$ captures all the variance, while projection along $\mathbf{w}_2$ captures none.

Key Takeaway

PCA identifies the directions of maximum variance, allowing dimensionality reduction without losing significant information.

.

Interpretation

Projecting the data onto the eigenvectors aligns it with the directions of maximal variance, effectively reducing dimensionality while preserving important patterns in the data.

Benefit

PCA simplifies visualization, accelerates learning, reduces noise, and helps mitigate the curse of dimensionality by focusing on the most informative directions in the data.

Step 5: Interpretation

- The first principal component captures the overall trend in the data, which in this example corresponds to the increasing pattern in both features.
- The second component, with zero eigenvalue, indicates no additional variance along the orthogonal direction.
- By retaining only the first component, we reduce the dataset from two dimensions to one dimension without significant loss of information.

Step 6: Practical Applications

- Data visualization in 2D or 3D scatter plots.
- Preprocessing before clustering or classification to reduce noise and improve algorithm performance.
- Compression of high-dimensional data in computer vision, bioinformatics, and finance.

Key Insight

PCA is a powerful tool to uncover the underlying structure of high-dimensional data. By projecting data onto the directions of maximum variance, PCA enables simpler, faster, and more interpretable analysis.

Python Implementation: Principal Component Analysis (PCA)

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Step 1: Sample 2D dataset
X = np.array([
    [2, 1],
    [3, 2],
    [4, 3]
])

# Step 2: Apply PCA
pca = PCA(n_components=1)  # Reduce to 1 dimension
X_pca = pca.fit_transform(X)

# Step 3: Display results
print("Original Data:\n", X)
print("Explained Variance Ratio:", pca.
    explained_variance_ratio_)
print("Principal Components:\n", pca.components_)
print("Transformed Data (1D Projection):\n", X_pca)

# Step 4: Visualization
plt.figure(figsize=(6,4))
plt.scatter(X[:,0], X[:,1], color='blue', label='Original
    Data')
plt.scatter(X_pca[:,0], np.zeros_like(X_pca), color='red',
    label='PCA Projection', s=100)
for i in range(X.shape[0]):
    plt.plot([X[i,0], X_pca[i,0]], [X[i,1], 0], 'k--', alpha
    =0.5)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('PCA: Projection onto First Principal Component')
plt.legend()
plt.grid(True)
```

```
plt.show()
```

Listing 3.2: PCA on Sample 2D Data

Explanation of Python Code

- We define a simple 2D dataset X with three samples.
- PCA is applied using `sklearn.decomposition.PCA` to reduce the data to 1 dimension.
- The `explained_variance_ratio_` shows how much variance is preserved.
- We visualize the original data in blue and its 1D PCA projection in red.
- Dashed lines indicate how each point is projected onto the principal component.

Interpretation

Even in this small example, PCA successfully identifies the direction along which data varies the most and projects points onto this line. This reduces dimensionality while preserving maximum variance, allowing simpler analysis, visualization, and preprocessing for downstream tasks.

3.3 Density Estimation

Density estimation is a core problem in unsupervised learning that focuses on modeling the underlying probability distribution of a dataset. Rather than grouping data into discrete clusters, density estimation aims to describe how data points are distributed across the feature space.

In the absence of labeled data, density estimation provides a probabilistic view of the data-generating process by estimating the probability density function (PDF) from observed samples.

Formally, given a dataset $\{x_1, x_2, \ldots, x_n\}$ drawn independently from an unknown distribution $p(x)$, the goal of density estimation is to construct an estimate $\hat{p}(x)$ such that:

- Regions with many data points have high probability density.
- Sparse regions correspond to low probability density.

Intuition

If data points frequently appear in certain regions of the feature space, those regions are likely to be more probable. Density estimation captures this intuition by assigning higher probability values to dense areas and lower values elsewhere.

Figure 3.3 illustrates the idea of density estimation for one-dimensional data, where peaks in the density function correspond to regions containing many samples.

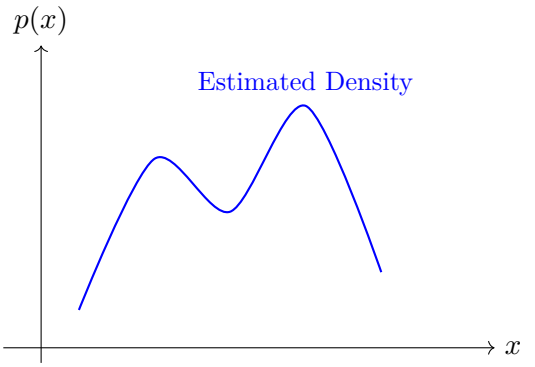


Figure 3.3: Conceptual illustration of density estimation. Peaks in the estimated probability density function correspond to regions where data points are concentrated.

Density estimation plays a crucial role in many machine learning tasks, including anomaly detection, data generation, probabilistic inference, and clustering. In fact, many clustering algorithms can be interpreted as implicitly performing density estimation.

Types of Density Estimation

Density estimation methods are generally categorized into two main types:

- **Parametric density estimation**: Assumes a specific functional form for the distribution (e.g., Gaussian).
- **Non-parametric density estimation**: Makes minimal assumptions about the shape of the distribution.

Each approach offers different trade-offs between flexibility, interpretability, and computational complexity.

Example: Gaussian Density Estimation

Gaussian density estimation is a parametric approach that assumes data is generated from a normal (Gaussian) distribution. In the one-dimensional case, the probability density function of a Gaussian distribution is given by:

$$p(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

where:

- μ is the mean of the distribution,
- σ^2 is the variance.

Real-World Example: Sensor Measurements

Gaussian density estimation is commonly used to model noise and variability in sensor data such as temperature, voltage, or current measurements.

Step 1: Dataset Consider a one-dimensional dataset representing daily temperature measurements (in degrees Celsius):

$$X = \{18, 19, 20, 21, 22, 23\}$$

These observations are assumed to be samples drawn from an underlying Gaussian distribution.

Step 2: Parameter Estimation The parameters of the Gaussian distribution are estimated using maximum likelihood estimation (MLE).

The estimated mean is:

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

The estimated variance is:

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu})^2$$

These parameters define the estimated probability density function $\hat{p}(x)$.

Step 3: Interpreting the Density Once the density is estimated:

- Values near the mean have high probability.
- Extreme values in the tails have low probability.

This probabilistic interpretation allows us to reason about how likely a new observation is under the learned model.

Interpretation

If a new temperature measurement has very low probability under the estimated density, it may indicate an abnormal or anomalous event.

Non-Parametric Density Estimation: Kernel Density Estimation

Kernel Density Estimation (KDE) is a widely used non-parametric method that estimates the density function directly from the data without assuming a specific distribution.

The KDE estimate is defined as:

$$\hat{p}(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

where:

- $K(\cdot)$ is the kernel function (commonly Gaussian),
- h is the bandwidth parameter controlling smoothness,
- n is the number of data points.

A small bandwidth leads to a highly detailed density estimate, while a large bandwidth produces a smoother curve.

Key Insight

Density estimation provides a probabilistic understanding of data distribution, enabling tasks such as anomaly detection, generative modeling, and uncertainty quantification without labeled data.

Python Implementation: Kernel Density Estimation

The following Python example demonstrates Kernel Density Estimation using the `scikit-learn` library.

```python
from sklearn.neighbors import KernelDensity
import numpy as np

# One-dimensional data
X = np.array([18, 19, 20, 21, 22, 23]).reshape(-1, 1)

# Initialize KDE model
kde = KernelDensity(kernel='gaussian', bandwidth=1.0)

# Fit model
kde.fit(X)

# Evaluate log-density
log_density = kde.score_samples(X)

print("Log Density Estimates:", log_density)
```

Listing 3.3: Kernel Density Estimation Example

The resulting density values quantify how likely each observation is under the estimated distribution, enabling probabilistic reasoning about unseen data.

3.4 Anomaly Detection

Anomaly detection is a key task in unsupervised learning that focuses on identifying observations which significantly deviate from the expected or normal patterns in a dataset. These unusual observations, also called outliers, often contain critical information about potential risks, failures, or fraudulent activities.

Key Insight

Anomalies are rare events that deviate from the typical distribution of data and often indicate important underlying phenomena such as fraud, failures, or intrusions.

Example Applications

- **Fraud Detection:** Detecting unusual transactions in banking or e-commerce that may indicate fraudulent activity.
- **Network Intrusion Detection:** Identifying suspicious patterns in network traffic that could signify cyber attacks.
- **Equipment Failure Monitoring:** Recognizing unusual sensor readings that indicate possible malfunction or failure in industrial systems.

- **Medical Diagnostics:** Detecting abnormal patterns in patient data for early diagnosis of diseases.

Key Concepts

- **Normal Behavior:** Observations that conform to the majority of data patterns.
- **Anomalies/Outliers:** Observations that do not follow common patterns and lie far from the bulk of the data.
- **Distance-based Detection:** Points far from cluster centers or neighbors are flagged as anomalies.
- **Density-based Detection:** Points in low-density regions are treated as anomalies (e.g., using Local Outlier Factor, LOF).
- **Reconstruction-based Detection:** Unusual points cause high reconstruction errors in models like autoencoders.

Numerical Illustration:: Z-score Based Anomaly Detection Consider a simple dataset of daily transaction amounts in arbitrary units:

$$X = [50, 52, 49, 51, 48, 300]$$

Step-by-Step Detection

Step 1: Compute Mean and Standard Deviation

$$\mu = \frac{50 + 52 + 49 + 51 + 48 + 300}{6} = 91.67, \quad \sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{n}} \approx 103.06$$

Step 2: Compute Z-scores

$$z_i = \frac{x_i - \mu}{\sigma}$$

$$z = [-0.40, -0.38, -0.42, -0.39, -0.43, 2.01]$$

Step 3: Identify Anomalies A common threshold is $|z| > 2$. Therefore, the last value 300 is flagged as an anomaly.
Interpretation: The unusually high transaction of 300 units stands out from the normal transactions ranging around 48–52.

```python
import numpy as np

# Sample data
X = np.array([50, 52, 49, 51, 48, 300])

# Compute mean and standard deviation
mu = np.mean(X)
sigma = np.std(X)

# Compute Z-scores
z_scores = (X - mu) / sigma

# Identify anomalies
threshold = 2
anomalies = X[np.abs(z_scores) > threshold]

print("Z-scores:", z_scores)
print("Anomalous values:", anomalies)
```

Listing 3.4: Z-score Based Anomaly Detection

Key Takeaways

- Anomalies often reveal critical and actionable insights.
- Methods like Z-score, distance-based, density-based, and reconstruction-based detection provide multiple ways to capture unusual behavior.
- Careful choice of thresholds and domain knowledge are crucial for effective anomaly detection.

3.5 Real-World Applications of Unsupervised Learning

Practical Importance of Unsupervised Learning

Unsupervised learning allows extracting meaningful patterns from raw, unlabeled data, enabling data-driven decisions in diverse domains.

3.5.1 Example: Market Segmentation with K-Means

Suppose we have customer spending data for two product categories: electronics and groceries. The dataset consists of four customers:

$$X = \begin{bmatrix} 200 & 300 \\ 220 & 310 \\ 150 & 100 \\ 160 & 120 \end{bmatrix}$$

Here, the first column represents annual spending on electronics, and the second column represents annual spending on groceries. Our goal is to automatically group these customers into two segments based on their purchasing behavior using **K-Means clustering**.

Step 1: Understanding the Data - Customers 1 and 2 have higher spending in both categories. - Customers 3 and 4 spend less overall. - Intuitively, we can expect two clusters: high-spending and low-spending customers.

Step 2: Applying K-Means Clustering K-Means algorithm works as follows:

1. Randomly initialize K cluster centroids (here $K = 2$).
2. Assign each data point to the nearest centroid.
3. Update centroids as the mean of assigned points.
4. Repeat steps 2-3 until convergence (no change in cluster assignments).

Step 3: Interpreting the Results After applying K-Means, each customer is assigned to a cluster, and the centroids represent the "average customer" in each segment.

```
from sklearn.cluster import KMeans
import numpy as np

# Customer spending data: [Electronics, Groceries]
X = np.array([
    [200, 300],
    [220, 310],
    [150, 100],
    [160, 120]
```

```
])

# Initialize K-Means with 2 clusters
kmeans = KMeans(n_clusters=2, random_state=42)
kmeans.fit(X)

# Cluster assignments
labels = kmeans.labels_
centroids = kmeans.cluster_centers_

print("Cluster labels:", labels)
print("Cluster centroids:\n", centroids)
```

Listing 3.5: K-Means Clustering for Customer Segmentation

Step 4: Analysis of Results Suppose the output is:

$$\text{Cluster Labels} = [0, 0, 1, 1], \quad \text{Centroids} = \begin{bmatrix} 210 & 305 \\ 155 & 110 \end{bmatrix}$$

Cluster 0 contains customers who spend more on both categories. Cluster 1 contains customers with lower spending. This segmentation can guide marketing strategies, e.g., premium offers for Cluster 0 and discount campaigns for Cluster 1.

3.5.2 Document and Topic Modeling

Topic modeling is an unsupervised learning technique used to uncover hidden thematic structures in large collections of documents. Each document can contain multiple topics, and each topic is represented by a distribution over words. One popular method is **Latent Dirichlet Allocation (LDA)**, which probabilistically assigns topics to documents and words to topics.

Key Insight

Topic modeling helps in summarizing, organizing, and understanding large text datasets by revealing hidden patterns and themes without requiring labeled data.

Example Consider a small text corpus consisting of 3 short documents and a vocabulary of 3 distinct words. Each document can be represented numerically by a **document-term matrix**, where each entry indicates the frequency of a particular word in a given document. For our example, the matrix is:

$$X = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

Our goal is to uncover 2 **latent topics** from this corpus using **Latent Dirichlet Allocation (LDA)**.

Latent Dirichlet Allocation (LDA): Concept

Latent Dirichlet Allocation (LDA) is a popular unsupervised learning technique used in **topic modeling**. It aims to uncover hidden thematic structures in a large collection of documents without requiring any labeled data.

Core Ideas:

- Each document is considered a **mixture of multiple topics**.
- Each topic is represented as a **probability distribution over words** in the vocabulary.
- LDA uses a generative probabilistic model to assign words to topics based on observed frequencies, estimating both:
  1. The **document-topic distribution**: probability of each topic in each document.
  2. The **topic-word distribution**: probability of each word under each topic.

Intuition: If a document frequently mentions words like *apple, banana, orange*, it likely belongs to a "fruits" topic, while another document with words *car, bus, train* belongs to a "vehicles" topic. LDA discovers such patterns automatically.

Applications:

- Organizing large text corpora
- Recommender systems
- Search and information retrieval
- Content summarization

LDA is particularly useful because it can handle **large-scale unlabeled text data**, providing interpretable topics that summarize the main themes of the corpus.

LDA assumes that:

- Each document is a mixture of multiple topics.
- Each topic is characterized by a probability distribution over words.

Thus, applying LDA will produce two key results:

1. A **document-topic distribution**, which indicates the probability of each topic appearing in each document. For example, Document 1 may be composed of 70% Topic 1 and 30% Topic 2.
2. A **topic-word distribution**, which indicates the probability of each word appearing under each topic. For instance, Topic 1 may be mostly associated with words 1 and 2, whereas Topic 2 may emphasize words 2 and 3.

Even with this simple example, LDA demonstrates its ability to automatically discover hidden themes and structure in text data without requiring any labeled information.

```
from sklearn.decomposition import LatentDirichletAllocation
import numpy as np

# Document-term matrix (rows: documents, columns: words)
X = np.array([
    [2, 1, 0],
    [1, 0, 1],
    [0, 1, 2]
])

# Initialize LDA with 2 topics
lda = LatentDirichletAllocation(n_components=2, random_state
    =42)
lda.fit(X)

# Document-topic distribution (probability of each topic per
    document)
doc_topic = lda.transform(X)

# Topic-word distribution (probability of each word per
    topic)
topic_word = lda.components_

print("Document-topic distribution:\n", doc_topic)
print("Topic-word distribution:\n", topic_word)
```

Listing 3.6: LDA Topic Modeling Example

Interpretation

- **Document-topic distribution:** Each document is represented as a probability distribution over topics. For example, Document 1 may be 80% Topic 1 and 20% Topic 2.
- **Topic-word distribution:** Each topic is represented as a probability distribution over words. For example, Topic 1 may be more associated with words 1 and 2, while Topic 2 may be associated with words 2 and 3.

This example illustrates how LDA can automatically extract latent themes from text, even in a small dataset, without any manual labeling.

3.5.3 Image Compression and Feature Learning

Dimensionality reduction techniques, such as Principal Component Analysis (PCA), are widely used in image processing. The main idea is to learn a **compact representation** of an image, reducing storage requirements and computational cost, while preserving the most **important features** or patterns in the data.

Intuition: Image Compression with PCA

An image can be thought of as a high-dimensional data matrix where each pixel is a feature. PCA identifies the directions (principal components) along which the image data varies the most. By keeping only the top principal components, we can represent the image with fewer numbers while retaining most of the important visual information.

Example Consider a simple 2×2 grayscale image represented by pixel intensities:

$$X = \begin{bmatrix} 100 & 150 \\ 200 & 250 \end{bmatrix}$$

Each row can be considered as an observation and each column as a feature. PCA can reduce this 2-dimensional data to a 1-dimensional representation while preserving the maximum variance.

Step 1: Center the Data Subtract the mean of each column:

$$\bar{X} = \begin{bmatrix} 100 - 150 & 150 - 200 \\ 200 - 150 & 250 - 200 \end{bmatrix} = \begin{bmatrix} -50 & -50 \\ 50 & 50 \end{bmatrix}$$

Step 2: Compute Covariance Matrix

$$\Sigma = \frac{1}{n}\bar{X}^\top \bar{X} = \frac{1}{2}\begin{bmatrix} (-50)^2 + 50^2 & (-50)(-50) + 50 \cdot 50 \\ (-50)(-50) + 50 \cdot 50 & (-50)^2 + 50^2 \end{bmatrix} = \begin{bmatrix} 2500 & 2500 \\ 2500 & 2500 \end{bmatrix}$$

Step 3: Eigen Decomposition The eigenvectors of Σ indicate the directions of maximum variance:

$$\lambda_1 = 5000, \quad \mathbf{w}_1 = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \lambda_2 = 0, \quad \mathbf{w}_2 = \frac{1}{\sqrt{2}}\begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

We select the top eigenvector $\mathbf{w}_1$ for the reduced representation.

Step 4: Projection onto Principal Component

$$X_{\text{reduced}} = \bar{X}\mathbf{w}_1 = \begin{bmatrix} -50 & -50 \\ 50 & 50 \end{bmatrix} \cdot \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -70.71 \\ 70.71 \end{bmatrix}$$

Step 5: Reconstruction (Optional) The original image can be approximately reconstructed:

$$X_{\text{reconstructed}} = X_{\text{reduced}}\mathbf{w}_1^\top + \text{mean} = \begin{bmatrix} -70.71 \\ 70.71 \end{bmatrix} \cdot \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \end{bmatrix} + \begin{bmatrix} 150 & 200 \end{bmatrix} \approx \begin{bmatrix} 100 & 150 \\ 200 & 250 \end{bmatrix}$$

```python
from sklearn.decomposition import PCA
import numpy as np

# 2x2 image pixel intensities
X = np.array([
    [100, 150],
    [200, 250]
])

# Apply PCA to reduce to 1D
pca = PCA(n_components=1)
X_reduced = pca.fit_transform(X)

# Reconstruct approximate original image
X_reconstructed = pca.inverse_transform(X_reduced)

print("Reduced representation:\n", X_reduced)
print("Reconstructed image:\n", X_reconstructed)
```

Listing 3.7: PCA for Image Compression

Key Takeaways

- PCA identifies directions of maximum variance in the image.
- Reducing dimensionality compresses the image while preserving essential visual features.
- Reconstruction may lose some information but retains the overall structure.

```
from sklearn.decomposition import PCA
from skimage.io import imread
import numpy as np

# Read the image file (grayscale)
# Make sure to replace 'image.jpg' with your actual image file path
image = imread('image.jpg', as_gray=True)

# Convert image to 2D array of pixel intensities
X = np.array(image, dtype=float)

print("Original image shape:", X.shape)

# Flatten the image into 2D matrix (rows: pixels, columns: features)
# For grayscale, each row can represent a pixel, each column a feature
X_flat = X.reshape(X.shape[0], X.shape[1])

# Apply PCA to reduce dimensions (e.g., keep top 50 components)
pca = PCA(n_components=50)
X_reduced = pca.fit_transform(X_flat)

# Reconstruct approximate image from reduced representation
X_reconstructed = pca.inverse_transform(X_reduced)

# Reshape back to original image shape
X_reconstructed = X_reconstructed.reshape(X.shape)

print("Reduced representation shape:", X_reduced.shape)
print("Reconstructed image shape:", X_reconstructed.shape)

# Optional: save reconstructed image using skimage
```

```
# from skimage.io import imsave
# imsave('reconstructed_image.jpg', X_reconstructed)
```

Listing 3.8: Image Compression using PCA on Actual Image

Explanation

- The image is read and converted into a 2D pixel array.
- PCA reduces the dimensionality of the pixel data while preserving the most significant features.
- The image is then reconstructed from the reduced representation, achieving compression.
- The reconstructed image retains the main visual features, though some fine details may be lost depending on the number of components kept.

3.5.4 Cybersecurity and Intrusion Detection

Unsupervised learning plays a crucial role in modern cybersecurity systems by enabling the detection of abnormal or suspicious behavior in network traffic, system logs, sensor data, and user activity patterns. Unlike supervised learning approaches, which rely on labeled datasets containing known attack signatures, unsupervised techniques operate without prior knowledge of attack types. This characteristic makes them particularly suitable for real-world cybersecurity environments, where labeled intrusion data is scarce, incomplete, or rapidly evolving due to the emergence of new and sophisticated attack vectors.

In cyber–physical systems such as smart grids, intrusion detection is a critical requirement to ensure system reliability, data integrity, and operational resilience. Cyberattacks such as Denial-of-Service (DoS), false data injection (FDI), malware propagation, and insider threats often manifest as subtle deviations from normal system behavior. Unsupervised learning algorithms, including clustering, density estimation, and statistical anomaly detection methods, can effectively identify such deviations by learning the underlying structure of normal operational data.

One of the most common applications of unsupervised learning in cybersecurity is anomaly detection. In this context, the system is trained using historical data that primarily represents normal behavior. Any future observation that significantly deviates from this learned baseline is considered

potentially malicious and flagged for further analysis. This approach is especially valuable in dynamic environments where attack patterns continuously change, rendering traditional rule-based or signature-based detection mechanisms less effective.

Example Consider a simple example involving network packet monitoring, where packet sizes (in bytes) are recorded over time:

$$X = [50, 52, 48, 51, 300, 49, 53, 50]$$

In this dataset, most packet sizes are clustered around 50 bytes, representing normal traffic behavior. However, the packet with a size of 300 bytes is significantly larger than the others. Such an abnormal value may indicate a potential cyber intrusion, such as a Denial-of-Service (DoS) attack, packet flooding, or a malformed packet attempting to exploit system vulnerabilities.

Step 1: Compute Z-Scores A common statistical approach for anomaly detection is Z-score normalization, which measures how far each data point deviates from the mean of the dataset. The Z-score for each observation is computed as:

$$z_i = \frac{x_i - \mu}{\sigma}$$

where μ represents the mean of the dataset and σ denotes the standard deviation. Z-score normalization converts the data into a standardized scale, allowing direct comparison of deviations regardless of the original units or magnitude.

Step 2: Identify Anomalies After computing the Z-scores, a threshold-based decision rule is applied to identify anomalies. Typically, data points with $|z_i| > 2$ or $|z_i| > 3$ are considered statistically significant outliers. In this example, the packet size of 300 bytes yields a very high Z-score compared to the rest of the data, thereby exceeding the predefined threshold and being flagged as an anomalous event.

Discussion and Practical Implications While Z-score–based anomaly detection is simple and computationally efficient, it assumes that the underlying data follows a normal distribution. In real-world cybersecurity applications, network traffic data may exhibit non-Gaussian behavior, temporal dependencies, and high dimensionality. Therefore, more advanced unsupervised learning techniques such as K-means clustering, Gaussian Mixture

Models (GMMs), Principal Component Analysis (PCA), Autoencoders, and Isolation Forests are often employed to improve detection accuracy.

In smart grid cybersecurity, these methods can be applied to monitor communication traffic between smart meters, substations, and control centers. By continuously learning normal operational patterns, unsupervised intrusion detection systems can provide early warnings of cyberattacks, reduce false alarms, and enhance the overall security posture of critical infrastructure. Consequently, unsupervised learning forms a foundational component of intelligent and adaptive cybersecurity frameworks in next-generation cyber–physical systems.

```python
import numpy as np

# Sample network packet sizes
X = np.array([50, 52, 48, 51, 300, 49, 53, 50])

# Compute mean and standard deviation
mu = np.mean(X)
sigma = np.std(X)

# Calculate Z-scores
z_scores = (X - mu) / sigma

# Define threshold for anomaly detection
threshold = 2

# Identify anomalous packet sizes
anomalies = X[np.abs(z_scores) > threshold]

print("Z-scores:", z_scores)
print("Anomalous packet sizes:", anomalies)
```

Listing 3.9: Z-score Based Anomaly Detection in Network Traffic

Step 3: Interpretation

- The Z-score method highlights packet sizes that deviate significantly from the typical range.
- In this example, the packet of size 300 bytes is correctly flagged as an anomaly.
- Such methods can be extended to multi-dimensional network features (e.g., packet size, duration, source/destination IP) for more sophisticated intrusion detection.

Key Takeaways

- Unsupervised learning uncovers hidden structures in unlabeled network data.
- Anomaly detection is critical for proactive cybersecurity and intrusion prevention.
- Simple statistical methods (e.g., Z-score) can be applied to single features, while advanced techniques like PCA, autoencoders, or clustering can handle multi-dimensional data.
- Python libraries such as `scikit-learn` and `numpy` facilitate rapid prototyping of unsupervised learning solutions for real-world cybersecurity tasks.

Limitations of Unsupervised Learning

Challenges and Limitations of Unsupervised Learning

Unsupervised learning is a powerful tool, but it comes with several limitations that practitioners should be aware of:

- **No Ground Truth:** Since the data is unlabeled, there is no definitive "correct answer" to validate the results. This makes model evaluation challenging.
- **Subjective Interpretation:** The output of clustering or dimensionality reduction algorithms often requires human interpretation. Different analysts may draw different conclusions from the same results.
- **Hyperparameter Sensitivity:** Methods like K-Means, DBSCAN, or LDA require careful selection of parameters (e.g., number of clusters, distance thresholds, or number of topics). Poor choices can lead to misleading results.
- **Scalability Issues:** Some algorithms may become computationally expensive with large datasets, requiring dimensionality reduction or approximate methods.
- **Noise Sensitivity:** Unsupervised models can be strongly influenced by outliers or noisy features, which may distort clustering or feature extraction.

Key Insight: Evaluating and interpreting unsupervised models often demands domain expertise and careful analysis. Supplementing with visualization, validation metrics (e.g., silhouette score, explained variance), or hybrid approaches can improve reliability.

Comparison with Supervised Learning

Supervised vs Unsupervised Learning

Aspect	Supervised	Unsupervised
Labeled data	Required	Not required
Learning objective	Prediction	Structure discovery
Evaluation	Straightforward	Challenging
Common tasks	Classification, Regression	Clustering, PCA

Summary

Unsupervised Learning Summary:

- Learns from unlabeled data
- Discovers hidden patterns and structure
- Includes clustering and dimensionality reduction
- Useful when labeled data is unavailable
- Complements supervised learning in real-world systems

Chapter 4

Reinforcement Learning

Reinforcement Learning (RL) is a way of training AI by using a system of **rewards** and **punishments**. Think of it like training a dog: when the dog performs a trick correctly, it gets a treat (positive reward); if it does something wrong, it gets no treat. Over time, the dog learns to perform the trick to maximize the number of treats it receives.

Key Idea: The Feedback Loop

In RL, an **Agent** (the AI) lives in an **Environment** (the world). It takes an **Action**, and the world gives it back two things: a new **State** (the new situation) and a **Reward** (how well it did).

4.1 Introduction

The Five Simple Parts of RL To understand RL, we only need to define five simple components:

- **State (s):** A "snapshot" of the current situation. For a chess AI, this is the position of all pieces on the board.
- **Action (a):** What the agent decides to do. For a robot, this could be "Move Forward" or "Turn Left."
- **Reward (r):** A number that tells the agent if it's doing well. $+10$ for winning a game, -10 for crashing into a wall.
- **Policy (π):** The agent's "strategy" or "brain." It is a rule that tells the agent: "If you see State s, you should do Action a."

- **Value (Q):** The agent's guess of the **long-term** benefit. It's not just about the immediate reward, but all the rewards it will get in the future.

Simple Logic: The Value Equation

The most important formula in RL is about updating our "guess" of how good an action is. We call this the **Q-Value**. The logic is:

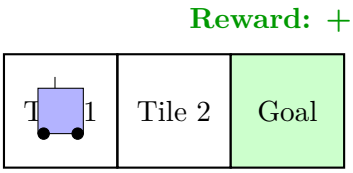
New Guess = Old Guess+Learning Rate×(Actual Reward−Old Guess)

In simple math:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma(\text{Best Future Guess}) - Q(s,a)]$$

Where: - α (Alpha) is how fast we learn from new info. - γ (Gamma) is how much we care about the future compared to now.

Example: A Robot in a Simple Room Imagine a robot in a room with three tiles. The robot starts at Tile 1 and wants to reach Tile 3 to get a big reward of +10.



The Setup: 1. The robot is at **Tile 1**. 2. It thinks moving to Tile 2 has a value of **0** ($Q = 0$). 3. It knows that once it gets to Tile 2, the value of the next step (to the Goal) is **10**.

The Action: The robot moves to Tile 2. It gets an immediate reward of **0** (it hasn't reached the goal yet).

The Update (Learning): We want to update the value of "Tile 1 → Tile 2". Let's use a Learning Rate (α) of 0.5 and a Discount (γ) of 0.9.

$$\text{Value Update} = 0 + 0.5 \times [0 + (0.9 \times 10) - 0]$$

$$\text{Value Update} = 0.5 \times 9 = 4.5$$

The Result: Now, the robot updates its memory. Next time it is at Tile 1, it knows that moving to Tile 2 is worth **4.5**. It has "learned" that this path leads to a reward later on!

How the AI Decides: Exploration vs. Exploitation A big problem for the AI is deciding between:

- **Exploitation:** Doing what it already knows works (Taking the path with the highest Q value).
- **Exploration:** Trying something completely new to see if there's a better reward elsewhere.

AI professionals usually solve this with the **Epsilon (ϵ) Strategy**: - Most of the time (90%), the AI is "Greedy" and takes the best path. - Sometimes (10%), the AI "Explores" and picks a random action.

aa

Concept	Simple Definition	Video Game Analogy
Agent	The Player	Mario
Environment	The Level	The obstacles and enemies
State	Current Screen	Where Mario is standing
Action	Controller Input	Pressing the 'Jump' button
Reward	Points / Health	Collecting a coin or dying

Table 4.1: Key concepts of Reinforcement Learning explained through gaming.

By repeating this loop millions of times, the AI builds a "Value Map" of the entire world, allowing it to navigate perfectly and achieve its goals.

4.1.1 Markov Decision Process (MDP)

Reinforcement learning problems are commonly modeled as a **Markov Decision Process (MDP)**, defined by the tuple:

$$\langle S, A, P, R, \gamma \rangle$$

where:

- S is the set of states,
- A is the set of actions,

- P defines state transition probabilities,
- R is the reward function,
- $\gamma \in [0, 1]$ is the discount factor.

4.1.2 Learning Mechanisms

The agent improves its policy using reinforcement learning algorithms such as Q-learning, SARSA, and policy gradient methods. These algorithms update value estimates or policies based on observed rewards and transitions, gradually converging to optimal behavior.

Final Insight: Reinforcement learning equips agents with the ability to make optimal decisions in complex, dynamic environments, forming the foundation of modern robotics, autonomous systems, and game-playing AI.

Comparative Overview of Learning methods

The selection of a learning method is dictated by the problem's architecture, the availability of labeled datasets, and the mechanism of feedback. While each method serves a distinct purpose, modern artificial intelligence often utilizes hybrid approaches to solve multifaceted tasks.

Table 4.2: Comparison of Primary Machine Learning methods

method	Data Type	Primary Goal	Common Algorithms
Supervised	Labeled	Mapping Inputs to Outputs	Linear Regression, SVM, Random Forest, CNNs
Unsupervised	Unlabeled	Finding Hidden Structure	K-means, PCA, Gaussian Mixture Models
Reinforcement	Environmental Feedback	Policy Optimization	Q-learning, PPO, Deep Q-Networks (DQN)

method Selection Note

The "No Free Lunch" theorem implies that no single method is superior for every problem. Supervised learning excels in classification, Unsupervised in discovery, and Reinforcement Learning in sequential decision-making.

Understanding learning methods is crucial for designing effective AI systems. Supervised learning is commonly used in predictive modeling tasks such as stock price prediction, spam detection, and medical diagnostics. Unsupervised learning is ideal for market segmentation, anomaly detection, and data compression. Reinforcement learning is applied in robotics, autonomous vehicles, game AI, and adaptive control systems. Moreover, hybrid methods combining supervised, unsupervised, and reinforcement learning are increasingly being employed for complex real-world problems, such as self-driving cars and intelligent recommendation systems. Machine learning provides AI systems with the ability to adapt, generalize, and make intelligent decisions. The selection of a suitable learning method lays the foundation for model design, training, and deployment, enabling systems to learn from data, interactions, and experiences in an efficient and scalable manner.

4.2 Autoencoders

What is an Autoencoder? (In Simple Words)

An **Autoencoder** is a type of neural network that learns by trying to **reproduce its own input**. Instead of learning from labeled answers, it learns patterns by observing the data itself.

In supervised learning, a model is trained using input–output pairs:

$$\text{Input} \rightarrow \text{Label}$$

However, in an autoencoder, the learning process is different:

$$\text{Input} \rightarrow \text{Same Input}$$

At first, this may sound useless. But the key idea is that the network is forced to pass the data through a **small hidden layer**. This forces it to learn only the **most important information** and ignore unnecessary details.

Example: Packing a Suitcase

Imagine you are going on a trip and have a very **large suitcase full of clothes**. Now the airline gives you a **small bag** and says:

> "You must fit all important items into this small bag."

To do this, you:

- Remove unnecessary items
- Fold clothes efficiently
- Keep only what truly matters

Later, when you reach your destination, you unpack the small bag and try to **recreate your original clothing setup** as best as possible.

- Packing → **Encoder**
- Small bag → **Latent Space**
- Unpacking → **Decoder**

This is exactly how an autoencoder works.

Basic Architecture of an Autoencoder

An autoencoder has three main parts:

1. **Input Layer** – original data
2. **Latent (Compressed) Layer** – important features only
3. **Output Layer** – reconstructed data

Figure 4.1: Simple working of an Autoencoder

Step-by-Step Explanation

1. Encoder (Compression) The encoder takes the original data and converts it into a smaller form.

$$\mathbf{z} = f(\mathbf{x})$$

For example:

- An image of $28 \times 28 = 784$ pixels
- Reduced to only 20 numbers

This step removes unnecessary details while keeping meaningful information.

2. Latent Space (Important Information) This is the **heart of the autoencoder**. It stores:

- Shape information
- General patterns
- Core structure of the data

If this layer is too large, the network memorizes the data. If it is small, the network learns **useful features**.

3. Decoder (Reconstruction) The decoder tries to rebuild the original input from the compressed data:

$$\hat{\mathbf{x}} = g(\mathbf{z})$$

The goal is to make the output as close as possible to the input.

How Does the Network Learn?

The autoencoder measures how different the reconstructed output is from the original input.

$$\text{Error} = \|\mathbf{x} - \hat{\mathbf{x}}\|^2$$

If the error is large:

- The network adjusts its weights

If the error is small:

- The network has learned good features

This process continues until reconstruction becomes accurate.

Example

Suppose we have a tiny black-and-white image:

- Original pixels: 16 values
- After encoding: only 3 numbers

$$\mathbf{x} \in \mathbb{R}^{16} \quad \rightarrow \quad \mathbf{z} \in \mathbb{R}^{3}$$

Even with only 3 numbers, the decoder can recreate a very similar image.

This proves that most images contain **redundant information**. Autoencoders learn the essence of the data.

Autoencoders are used for

- **Image Denoising:** Removing noise from old photos
- **Data Compression:** Reducing file size
- **Anomaly Detection:** Detecting faulty machines or fraud
- **Feature Learning:** Preparing data for deep learning models

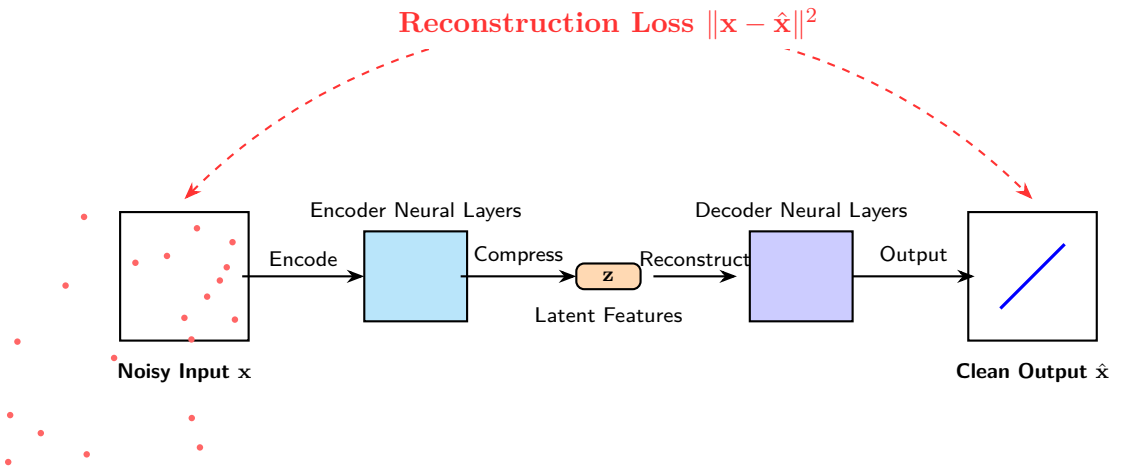


Figure 4.2: Internal working of an autoencoder for noise removal

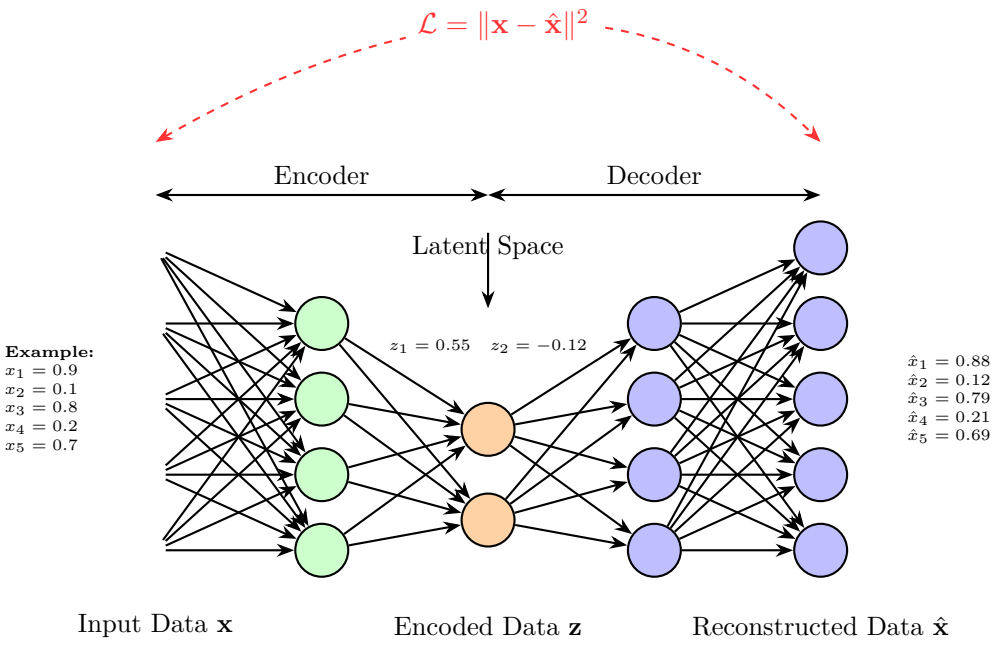


Figure 4.3: Autoencoder architecture for a simple Example

4.2.1 Step-by-Step Numerical Explanation of an Autoencoder

To clearly understand how an autoencoder works, we now explain the architecture shown in Figure **??** using a simple Example. The goal of an autoencoder is to learn a compact representation of input data and then reconstruct it with minimum error.

Step 1: Input Vector Representation

Consider an input data vector consisting of five numerical features:

$$\mathbf{x} = [0.9,\ 0.1,\ 0.8,\ 0.2,\ 0.7]^T$$

This vector is shown on the left side of the figure as five input neurons. In practice, these values may represent sensor readings, normalized measurements, or extracted features. At this stage, the data may contain redundancy or noise.

Each input neuron forwards its value to every neuron in the first encoder layer through weighted connections.

Step 2: Encoder Layer Computation (With Numerical Illustration:)

The encoder transforms the high-dimensional input vector into a smaller set of meaningful features. This transformation is achieved using a weighted sum followed by a nonlinear activation function. Mathematically, the encoder operation is defined as:

$$\mathbf{h} = f(\mathbf{W}_e\mathbf{x} + \mathbf{b}_e)$$

where:

- $\mathbf{W}_e$ is the encoder weight matrix,
- $\mathbf{b}_e$ is the bias vector,
- $f(\cdot)$ is a nonlinear activation function such as ReLU or sigmoid.

Weighted Sum Computation Consider the input vector from Step 1:

$$\mathbf{x} = [0.9,\ 0.1,\ 0.8,\ 0.2,\ 0.7]^T$$

Assume the encoder consists of **four hidden neurons**. Each hidden neuron computes a weighted sum of all five input values.

For illustration, let the encoder weight matrix and bias vector be:

$$\mathbf{W}_e = \begin{bmatrix} 0.4 & 0.1 & 0.3 & 0.2 & 0.5 \\ 0.2 & 0.6 & 0.1 & 0.3 & 0.4 \\ 0.5 & 0.2 & 0.4 & 0.1 & 0.3 \\ 0.3 & 0.4 & 0.2 & 0.5 & 0.1 \end{bmatrix}, \quad \mathbf{b}_e = \begin{bmatrix} 0.05 \\ -0.02 \\ 0.04 \\ 0.01 \end{bmatrix}$$

The pre-activation output of the encoder is computed as:

$$\mathbf{a} = \mathbf{W}_e\mathbf{x} + \mathbf{b}_e$$

For example, the first hidden neuron computes:

$$a_1 = (0.4)(0.9) + (0.1)(0.1) + (0.3)(0.8) + (0.2)(0.2) + (0.5)(0.7) + 0.05 = 1.03$$

Similarly, the remaining neurons produce:

$$\mathbf{a} \approx [1.03,\ 0.61,\ 0.87,\ 0.54]^T$$

Activation Function Next, a nonlinear activation function is applied. Assuming a sigmoid activation:

$$f(a) = \frac{1}{1 + e^{-a}}$$

The hidden layer output becomes:

$$\mathbf{h} = f(\mathbf{a}) \approx [0.74,\ 0.65,\ 0.70,\ 0.63]^T$$

These hidden neurons now contain compressed and filtered information derived from the original input.

Mapping to the Latent Space The hidden layer outputs are then projected into the **latent space** using another linear transformation:

$$\mathbf{z} = \mathbf{W}_z \mathbf{h} + \mathbf{b}_z$$

Assuming:

$$\mathbf{W}_z = \begin{bmatrix} 0.6 & 0.4 & 0.5 & 0.3 \\ -0.4 & 0.2 & -0.3 & 0.1 \end{bmatrix}, \quad \mathbf{b}_z = \begin{bmatrix} 0.02 \\ -0.01 \end{bmatrix}$$

The resulting latent representation is:

$$\mathbf{z} \approx [0.55,\ -0.12]^T$$

These are exactly the encoded values shown in Step 3.

Interpretation This encoder computation shows how:

- Multiple input features are combined through learned weights,
- Nonlinearity helps capture complex patterns,
- Dimensionality is reduced in stages,
- Only the most informative components survive in the latent space.

Thus, the encoder successfully compresses the five-dimensional input into a two-dimensional latent vector that retains the core structure of the data.

Step 3: Latent Space (Encoded Representation)

The most important part of the autoencoder is the **latent space**, also called the bottleneck layer. Here, the network compresses the information into only two values:

$$\mathbf{z} = [z_1,\ z_2]^T = [0.55,\ -0.12]^T$$

This drastic reduction from five dimensions to two forces the autoencoder to retain only the most essential features of the input. Redundant and noisy information is naturally discarded because the network has limited capacity.

The latent vector $\mathbf{z}$ represents a compact summary of the original data and is the output of the encoder.

Step 4: Decoder Reconstruction

Once the encoder has compressed the input into the latent vector, the decoder attempts to reconstruct the original input using only this compact representation. The decoder performs the inverse operation of the encoder by gradually expanding the latent features back to the original data dimension.

Mathematically, the decoder operation is defined as:

$$\hat{\mathbf{x}} = g(\mathbf{W}_d\mathbf{z} + \mathbf{b}_d)$$

where:

- $\mathbf{W}_d$ is the decoder weight matrix,
- $\mathbf{b}_d$ is the decoder bias vector,
- $g(\cdot)$ is a nonlinear activation function, typically the same as used in the encoder.

Latent Vector as Input From Step 3, the latent representation is:

$$\mathbf{z} = [0.55,\ -0.12]^T$$

This two-dimensional vector contains the most essential information extracted from the original five-dimensional input.

Decoder Hidden Layer Computation Assume the decoder first expands the latent vector into **four hidden neurons**. Let the decoder weight matrix and bias vector be:

$$\mathbf{W}_d^{(1)} = \begin{bmatrix} 0.5 & -0.3 \\ 0.4 & 0.2 \\ 0.6 & -0.1 \\ 0.3 & 0.4 \end{bmatrix}, \quad \mathbf{b}_d^{(1)} = \begin{bmatrix} 0.04 \\ 0.02 \\ 0.03 \\ 0.01 \end{bmatrix}$$

The pre-activation values of the decoder hidden layer are computed as:

$$\mathbf{a}_d = \mathbf{W}_d^{(1)}\mathbf{z} + \mathbf{b}_d^{(1)}$$

For example, the first hidden neuron computes:

$$a_{d1} = (0.5)(0.55) + (-0.3)(-0.12) + 0.04 = 0.35$$

Similarly, the remaining neurons produce:

$$\mathbf{a}_d \approx [0.35,\ 0.28,\ 0.36,\ 0.23]^T$$

Applying a sigmoid activation function:

$$\mathbf{h}_d = g(\mathbf{a}_d) \approx [0.59,\ 0.57,\ 0.59,\ 0.56]^T$$

Output Layer Reconstruction Next, the decoder maps the hidden layer back to the original five-dimensional space. Assume the output layer parameters are:

$$\mathbf{W}_d^{(2)} = \begin{bmatrix} 0.6 & 0.3 & 0.4 & 0.2 \\ 0.2 & 0.5 & 0.3 & 0.4 \\ 0.5 & 0.2 & 0.6 & 0.3 \\ 0.3 & 0.4 & 0.2 & 0.5 \\ 0.4 & 0.3 & 0.5 & 0.2 \end{bmatrix}, \quad \mathbf{b}_d^{(2)} = \begin{bmatrix} 0.02 \\ 0.01 \\ 0.03 \\ 0.02 \\ 0.01 \end{bmatrix}$$

The reconstructed output is computed as:

$$\hat{\mathbf{x}} = g(\mathbf{W}_d^{(2)}\mathbf{h}_d + \mathbf{b}_d^{(2)})$$

After applying the activation function, the reconstructed vector becomes:

$$\hat{\mathbf{x}} \approx [0.88,\ 0.12,\ 0.79,\ 0.21,\ 0.69]^T$$

Interpretation The reconstructed values are very close to the original input:

$$\mathbf{x} = [0.9,\ 0.1,\ 0.8,\ 0.2,\ 0.7]^T$$

This demonstrates that:

- The latent space retains sufficient information,

- The decoder successfully reverses the compression process,
- Minor differences occur due to dimensionality reduction and nonlinear activations.

Thus, the decoder completes the autoencoder cycle by transforming compact latent features back into a high-dimensional reconstruction of the original data.

Step 5: Reconstructed Output Vector

The output layer produces the reconstructed vector:

$$\hat{\mathbf{x}} = [0.88,\ 0.12,\ 0.79,\ 0.21,\ 0.69]^T$$

Notice that the reconstructed values are very close to the original input values. Small differences occur due to compression, activation nonlinearities, and learning limitations.

Step 6: Loss Function and Learning Objective

The autoencoder is trained by minimizing the reconstruction error between the input and output. A commonly used loss function is the mean squared error (MSE):

$$\mathcal{L} = \|\mathbf{x} - \hat{\mathbf{x}}\|^2$$

During training, backpropagation adjusts all weights and biases in both the encoder and decoder to minimize this loss over many examples.

Key Learning Insight

This Example demonstrates the core principle of autoencoders:

- The encoder **compresses** data into a low-dimensional latent space.
- The latent space captures the most informative features.
- The decoder **reconstructs** the original data using only these features.
- Noise and redundancy are not preserved during compression.

As a result, autoencoders are widely used for dimensionality reduction, noise removal, feature learning, and anomaly detection.

Autoencoders teach machines how to **understand data without labels**. By learning how to compress and reconstruct information, they discover patterns, remove noise, and create meaningful representations = a key skill for modern AI systems.

```python
# Import required libraries
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Step 1: Input vector
# 5 features as given in the example
X = np.array([[0.9, 0.1, 0.8, 0.2, 0.7]])

# Step 2: Define the autoencoder architecture
autoencoder = Sequential([
    Dense(4, activation='relu', input_shape=(5,)),  # Encoder layer: 5 -> 4
    Dense(2, activation='relu'),                    # Latent space (bottleneck): 4 -> 2
    Dense(4, activation='relu'),                    # Decoder intermediate layer: 2 -> 4
    Dense(5, activation='linear')                   # Output layer: 4 -> 5
])

# Step 3: Compile the model
autoencoder.compile(optimizer='adam', loss='mse')

# Step 4: Train the autoencoder
autoencoder.fit(X, X, epochs=500, verbose=0)

# Step 5: Extract encoded (latent) representation
encoder = Sequential(autoencoder.layers[:2])  # Encoder: first 2 layers
latent_vector = encoder.predict(X)

# Step 6: Reconstruct input from latent vector
reconstructed_X = autoencoder.predict(X)

# Step 7: Display results
print("Original Input Vector:\n", X)
print("\nLatent (Encoded) Vector:\n", latent_vector)
print("\nReconstructed Output Vector:\n", reconstructed_X)
```

Listing 4.1: Autoencoder Implementation for the 5-Dimensional Example

4.3 Self-Organizing Maps (SOMs): The Topology of Data

➢ Unsupervised Feature Mapping

Self-Organizing Maps (SOMs), also known as Kohonen Maps, are a class of **unsupervised neural networks** designed to project high-dimensional data onto a lower-dimensional (**typically 2D**) grid. Unlike traditional neural networks that minimize error via backpropagation, SOMs use **competitive learning** to preserve the **topological properties** of the input space.

The fundamental principle of an SOM is **topology preservation**:

> *Input vectors that are similar in the original high-dimensional space are mapped to the same or nearby neurons on the 2D grid, creating a "spatial landscape" of data relationships.*

The Mechanics of Competitive Learning:

The training of an SOM follows a "Winner-Take-All" philosophy. The process can be broken down into four distinct mathematical phases:

1. **Initialization:** Weights for each node on the grid are initialized randomly.

2. **Competition:** For each input vector $\mathbf{x}$, every node i calculates its Euclidean distance: $d_i = \|\mathbf{x} - \mathbf{w}_i\|$. The node with the smallest distance is the **Best Matching Unit (BMU)**.

3. **Cooperation:** The BMU determines a "neighborhood" of nearby nodes.

4. **Adaptation:** The weights of the BMU and its neighbors are updated:

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + \eta(t) \cdot L(i, \text{BMU}, t) \cdot [\mathbf{x} - \mathbf{w}_i(t)]$$

Case Study: Customer Segmentation in Retail

Consider a retail company analyzing customer behavior (Frequency, Spend, and Category Preference). The SOM maps these high-dimensional profiles onto a 2D grid where hidden structures become easy to interpret.

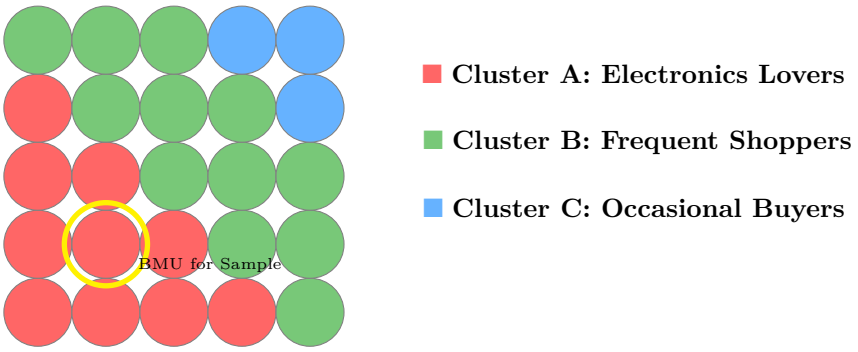


Figure 4.4: 2D Grid Representation of an SOM for Customer Segmentation

Numerical Interpretation:

Assume we have three-dimensional customer feature vectors:

ID	Freq.	Spend	Elec. Pref.	SOM Assignment
Cust 1	8	250	0.9	**Cluster A** (High Spend)
Cust 2	2	50	0.2	**Cluster C** (Occasional)
Cust 3	5	120	0.5	**Cluster B** (Consistent)

Strategic Insight: Marketing teams use these spatial distances to design targeted campaigns. If Cluster A is close to Cluster B, it suggests a natural "migration path" to convert consistent shoppers into high-value electronics lovers.

Final Interpretation The SOM grid preserves the **topological structure**: nodes that are close together represent customers with similar behavior, while distant nodes correspond to significantly different profiles. This visualization allows for complex data-driven business decisions at a glance.

4.4 Python Libraries for Machine Learning

The AI Ecosystem

Python has emerged as the industry standard for machine learning due to its simplicity and a rich ecosystem of libraries. These tools provide pre-built functions and optimized algorithms, allowing practitioners to focus on **architecture design** rather than low-level implementation.

The "Big Four" Core Libraries

NumPy: Numerical Computing Foundation NumPy is the backbone of scientific computing. It provides high-performance multi-dimensional arrays (`ndarrays`) and vectorized operations.

```
import numpy as np
X = np.array([[1,2], [3,4], [5,6]])
y = np.array([1,2,3])
# Compute pseudo-inverse for linear regression
w = np.linalg.pinv(X.T @ X) @ X.T @ y
```

Matrix Operations with NumPy

Pandas: Data Manipulation and Analysis Pandas introduces the `DataFrame`, a powerful 2D data structure that simplifies data cleaning, handling missing values, and SQL-like merging.

```
import pandas as pd
df = pd.read_csv('data.csv')
df.fillna(df.mean(), inplace=True) # Replace NaN with mean
```

Data Cleaning with Pandas

Matplotlib and Seaborn: Visualization Matplotlib provides low-level control for 2D plots, while Seaborn offers high-level statistical themes built on top of it.

Scikit-learn: The ML Toolbox Scikit-learn is the most comprehensive library for "classical" ML (regression, clustering, and preprocessing).

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

```

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2)
model = LogisticRegression().fit(X_train, y_train)
y_pred = model.predict(X_test)
```

Logistic Regression with Scikit-learn

4.4.1 Deep Learning Frameworks

For building complex neural networks, two frameworks dominate the field:

- **TensorFlow (Google):** Excellent for production deployment and large-scale industrial pipelines.
- **PyTorch (Meta):** Favored in research for its *dynamic computation graphs* and intuitive Pythonic feel.

Typical Machine Learning Workflow

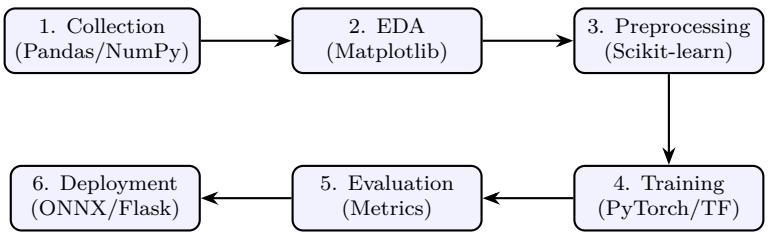


Figure 4.5: The end-to-end Machine Learning Pipeline in Python.

Concluding Perspective

Concluding Perspective

Python's ecosystem significantly accelerates the development of intelligent systems. By combining these specialized tools, practitioners can seamlessly implement end-to-end workflows for diverse applications, ranging from predictive analytics to advanced computer vision.

Part II

Artificial Neural Networks and Deep Learning

Chapter 5

Artificial Neural Networks

What is Prediction?

Prediction: The Essence of Artificial Intelligence

Prediction is the fundamental operation of artificial neural networks, enabling machines to transform inputs into meaningful outputs using learned internal computations.

Prediction refers to the process of producing an output for a given input based on prior learning. In artificial intelligence, prediction is not random guessing; rather, it is a structured and mathematical computation driven by trained parameters such as weights and biases.

At an intuitive level, prediction closely resembles human reasoning. When a human is asked a question, they internally think and then respond with an answer. Artificial neural networks follow the same conceptual flow.

$$\text{Input} \longrightarrow \text{Internal Processing} \longrightarrow \text{Output}$$

—

Human Thinking Analogy

“‘latex

This diagram highlights that intelligence=whether biological or artificial=depends on an internal transformation stage between input and output.

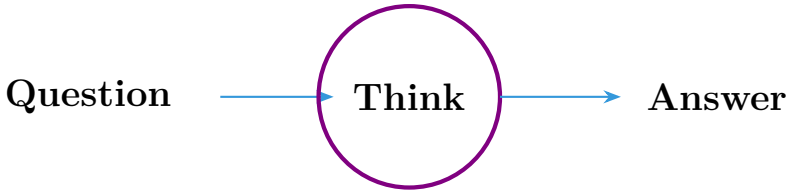


Figure 5.1: Human-style reasoning process

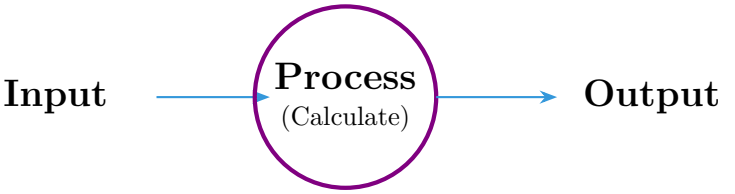


Figure 5.2: Prediction as a computational process in machines

Machine Prediction Perspective

In a neural network, this internal process consists of weighted summation, bias addition, and nonlinear activation.

Example of Prediction

Consider the simple arithmetic task: 2×5=?

A neural network does not inherently understand multiplication. Instead, it learns an internal computation that produces the correct output.

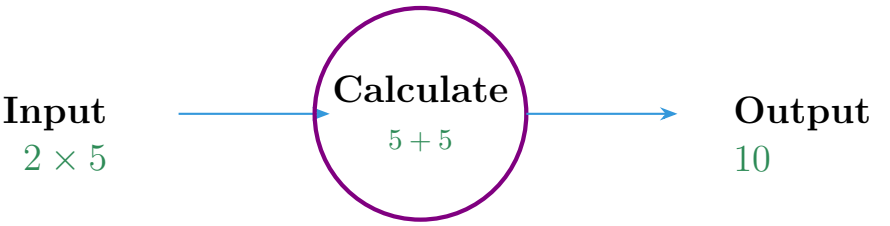


Figure 5.3: Prediction through learned internal computation

This example illustrates that prediction is achieved through internal processing rather than direct rule execution.

Why Prediction Matters

Why Prediction is Central

The true power of artificial intelligence lies in its ability to make accurate predictions on unseen data.

Prediction enables a wide range of applications, image recognition, speech processing, medical diagnosis, financial forecasting, and intelligent decision-making systems.

5.1 A Fundamental Predictive Model: Prediction, Error, and Learning Through Refinement

Overview

Prediction lies at the core of all intelligent systems. A machine does not *think* or *guess* in a human sense; instead, it executes an internal mathematical transformation whose behavior can be systematically improved through experience.
In this section, we study the simplest possible predictive model to understand how machines transform inputs into outputs, measure their mistakes, and gradually refine their internal parameters.

1. What is a Predictive Model?

A predictive model is a mathematical mechanism that:

- Receives one or more numerical inputs
- Processes them using adjustable internal parameters
- Produces an estimated output (prediction)

The defining characteristic of such a model is that its parameters are **learned from data**, not fixed in advance.

From a machine-learning perspective, prediction follows the same conceptual structure as human reasoning:

$$\text{Input} \longrightarrow \text{Internal Computation} \longrightarrow \text{Output}$$

Mathematically, the simplest predictive rule can be written as:

$$\hat{y} = w \times x$$

where:

- x is the input feature
- w is an adjustable weight (internal parameter)
- $\hat{y}$ is the predicted output

This equation represents a machine's *internal processing stage*.

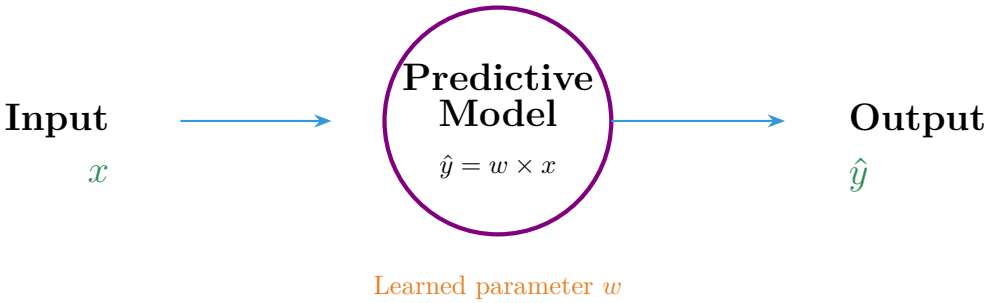


Figure 5.4: A predictive model as an internal mathematical transformation

Final Insight

Every artificial neural network=regardless of its size or complexity= is fundamentally a prediction engine that transforms inputs into outputs through learned internal computation.

2.Example: Estimating House Rent

Consider a real-world question:

Can a machine predict monthly house rent using house size?

We assume a simple linear relationship:

$$\text{Predicted Rent} = w \times \text{Area}$$

Initially, the machine does **not know** the correct value of w. It must learn it through repeated prediction and error correction.

Iteration 1: Initial Guess (Strong Under-estimation)

$$w = 8 \quad \Rightarrow \quad \hat{y} = 8 \times 50 = 400$$

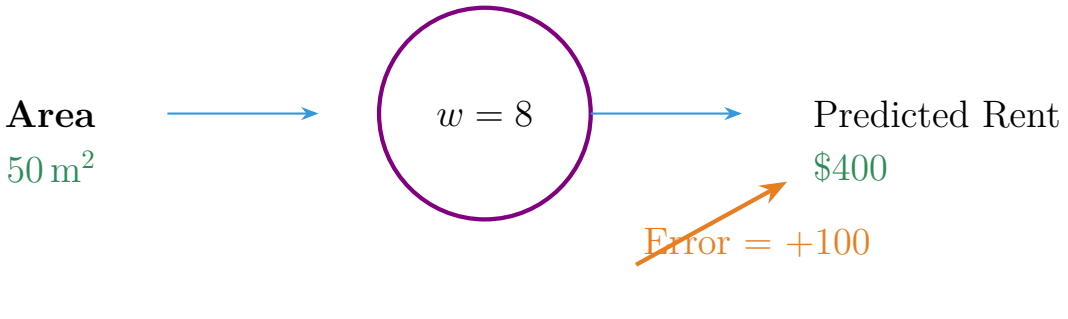


Figure 5.5: Large positive error due to low weight

The model underestimates the rent, producing a large positive error.

Iteration 2: Reduced Under-estimation

$$w = 9 \quad \Rightarrow \quad \hat{y} = 9 \times 50 = 450$$

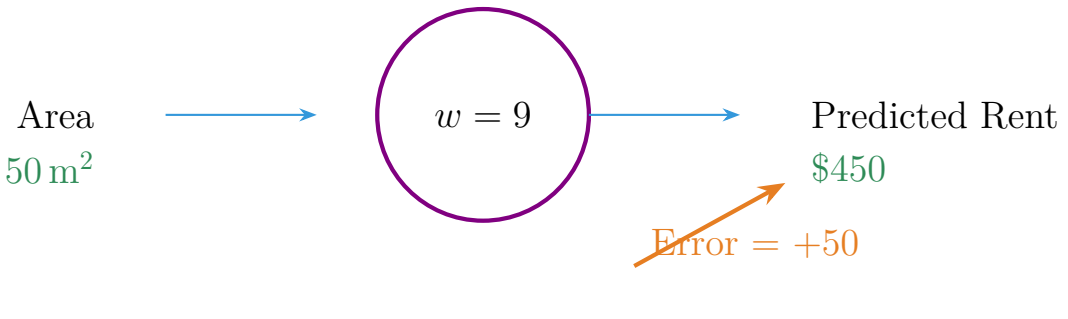


Figure 5.6: Error decreases as weight increases

The error magnitude is smaller, indicating progress toward the correct value.

Iteration 3: Near-Accurate Prediction

$$w = 9.5 \quad \Rightarrow \quad \hat{y} = 9.5 \times 50 = 475$$

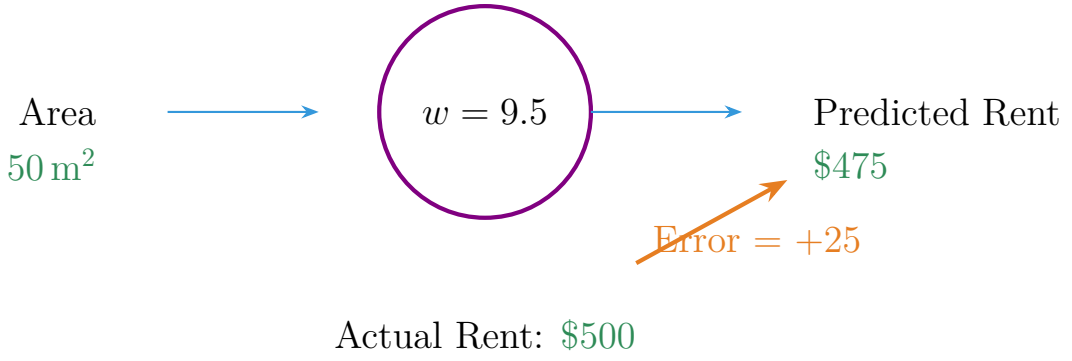


Figure 5.7: Prediction approaching the true value

Iteration 4: Correct Estimation

$$w = 10 \quad \Rightarrow \quad \hat{y} = 500$$

Perfect Prediction

The predicted rent exactly matches the actual rent.

Iteration 5: Over-estimation (Negative Error)

$$w = 11 \quad \Rightarrow \quad \hat{y} = 550$$

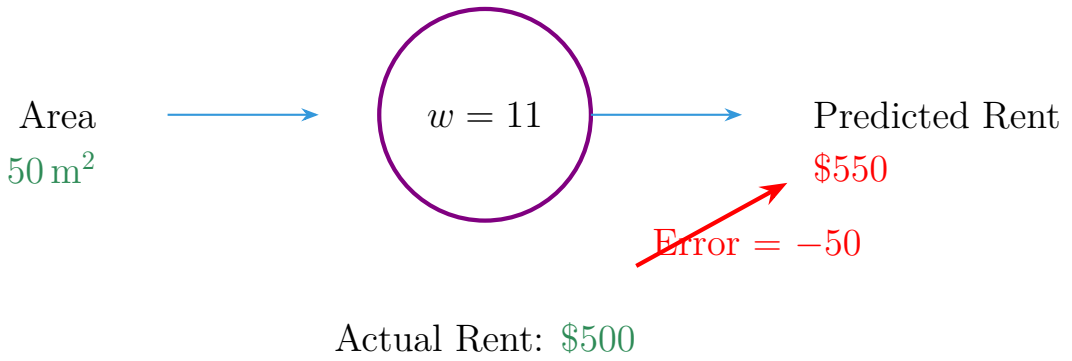


Figure 5.8: Negative error caused by over-estimation

The negative error signals that the weight has become too large.

Learning Insight

- Positive error ⇒ increase weight
- Negative error ⇒ decrease weight
- Learning proceeds through gradual refinement
- Accurate prediction emerges through iteration

Converged Prediction

Predicted Rent = Actual Rent

- Under-estimation produces **positive error**
- Over-estimation produces **negative error**
- Errors guide corrective updates
- Learning converges through oscillation and refinement

This iterative correction mechanism is the foundation of learning in linear models and artificial neural networks.

5.2 Drawing a Line Using the Slope–Intercept Form

Objective

This section demonstrates how to draw a straight line step by step using the slope–intercept form of a linear equation.

We are given the equation:

$$y = 2x + 3$$

and the range:

$$0 \leq x \leq 5$$

Step 1: Identify Slope and Intercept

The slope–intercept form of a line is:

$$y = mx + b$$

By comparison:

$$m = 2 \quad \text{and} \quad b = 3$$

- The slope $m = 2$ means the line rises by 2 units for every 1 unit increase in x.

- The intercept $b = 3$ means the line crosses the y-axis at $y = 3$.

Step 2: Create a Table of Values

Using the given range of x, we compute the corresponding values of y.

x	$y = 2x + 3$
0	3
1	5
2	7
3	9
4	11
5	13

Table 5.1: Computed points for the line $y = 2x + 3$

Each row in the table represents a point (x, y) on the line.

Step 3: Draw the Graph Paper Grid

We now draw a coordinate system with evenly spaced grid lines, similar to graph paper.

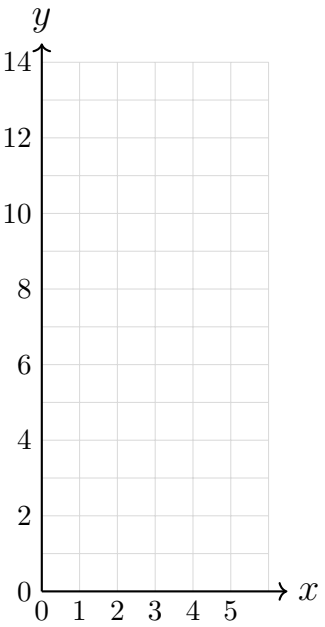


Figure 5.9: Graph-paper style coordinate system

Step 4: Plot Key Points

Using the table, we plot points that satisfy the equation.

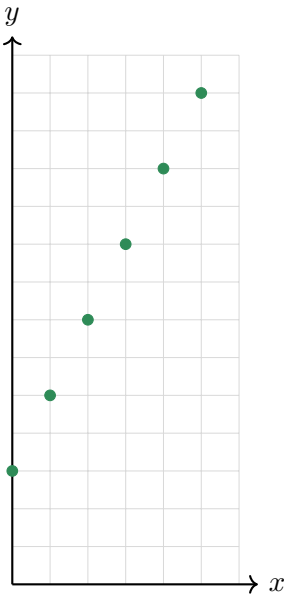


Figure 5.10: Points plotted using $y = 2x + 3$

Step 5: Draw the Straight Line

Finally, we draw a straight line passing through the plotted points.

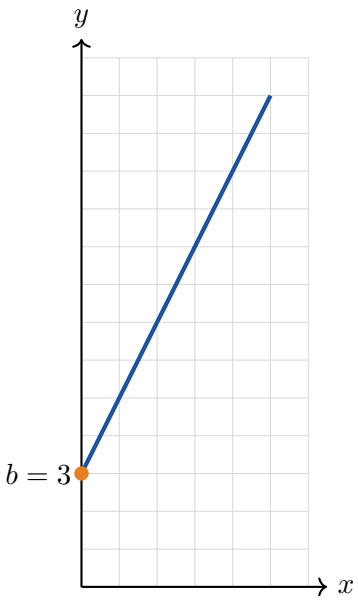


Figure 5.11: Line drawn using slope–intercept form

Key Insight

Final Insight

To draw a straight line using the slope–intercept form:

1. Identify the slope and intercept
2. Compute a small set of points
3. Plot the points on graph paper
4. Join them with a straight line

A Line Using the Slope-Intercept form with Zero Intercept (passing through origin)

Objective

This section demonstrates how to draw a straight line step by step using the slope–intercept form when the line passes through the origin.

We are given the linear equation:

$$y = 2x$$

with the input range:

$$0 \leq x \leq 5$$

Step 1: Identify Slope and Intercept

The slope–intercept form of a line is:

$$y = mx + b$$

For the given equation:

$$m = 2, \qquad b = 0$$

- The slope $m = 2$ means that for every increase of 1 unit in x, the value of y increases by 2 units.
- The intercept $b = 0$ indicates that the line passes through the origin.

Step 2: Construct a Table of Values

Using the specified range of x, we compute the corresponding values of y.

x	$y = 2x$
0	0
1	2
2	4
3	6
4	8
5	10

Table 5.2: Points generated from the equation $y = 2x$

Each row represents a point (x, y) lying on the line.

Step 3: Draw a Graph-Paper Style Grid

We now create a coordinate system with equal spacing on both axes, similar to standard graph paper.

Step 4: Plot the Points

Using the table of values, we plot the corresponding points.

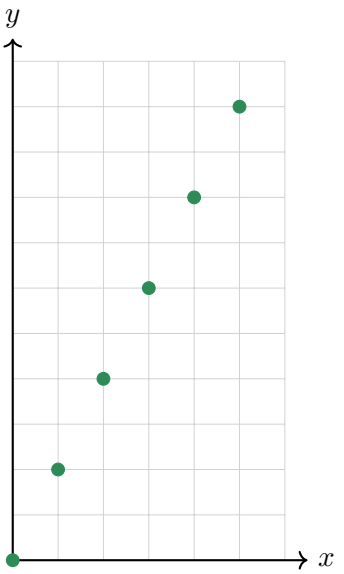


Figure 5.12: Points plotted for the line $y = 2x$

Step 5: Draw the Straight Line

Finally, we draw a straight line passing through all the plotted points.

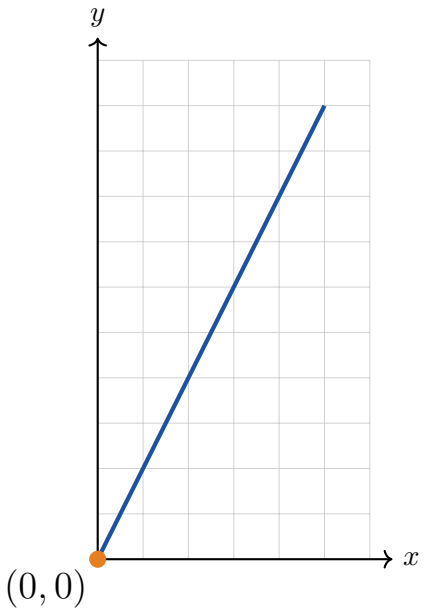


Figure 5.13: Line $y = 2x$ drawn from the origin

Key Insight

Final Insight

When the y-intercept is zero, the line always passes through the origin. The slope alone determines the direction and steepness of the line.

5.3 Training a Simple Linear Classifier

In this section, we demonstrate how a very simple linear classifier can be trained using a small number of labeled examples. Rather than immediately introducing abstract mathematical theory, we take an intuitive, example-driven approach. This allows us to gradually develop both understanding and confidence in how learning from data actually works.

Our goal is to train a linear classifier that can correctly distinguish between two types of **cell phones** based on their physical dimensions.

5.3.1 Problem Description

We consider the following classification task:

- **Compact Phones**

- **Large Phones**

Each phone is described using two numerical features:

1. **Width**
2. **Height**

By plotting width against height, each phone can be represented as a single point in a two-dimensional feature space. Our objective is to learn a straight line that separates compact phones from large phones.

Training data

To keep the example conceptually simple while still being realistic, we now consider a slightly richer set of labeled training examples. Instead of relying on only two phones, we use a small collection of nine examples. These examples represent *ground truth* = data points whose correct classifications are already known.

Using multiple examples allows the learning problem to better reflect real-world variability while remaining easy to visualize and analyze.

Table 5.3: Training Data for Cell Phone Classification

Example	Width (cm)	Height (cm)	Phone Type
1	3.0	1.0	Compact Phone
2	3.2	1.2	Compact Phone
3	2.8	1.1	Compact Phone
4	3.4	0.9	Compact Phone
5	1.0	3.0	Large Phone
6	1.2	3.2	Large Phone
7	0.9	2.8	Large Phone
8	1.1	3.4	Large Phone
9	1.3	3.1	Large Phone

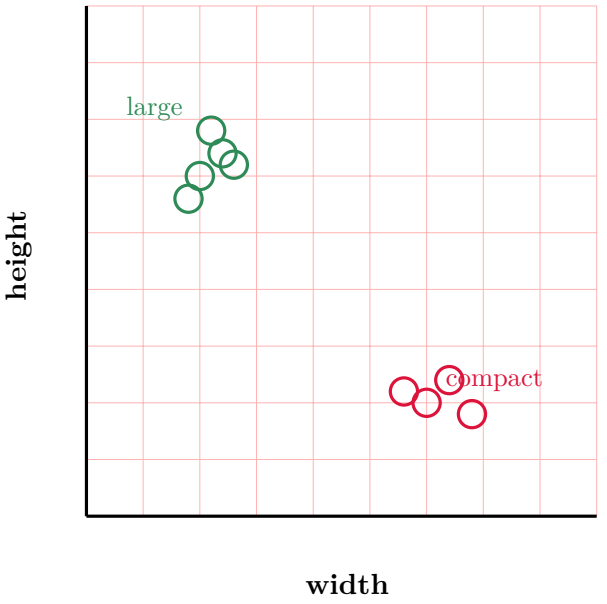
The first four training examples represent phones that are relatively wide but short. These devices are labeled as **compact phones**, reflecting their broader shape and reduced vertical length.

The remaining five examples represent phones that are taller and narrower. These are labeled as **large phones**, capturing the elongated design typically associated with modern large-screen devices.

Such labeled examples collectively form the **training data**. They provide structured information that allows the learning algorithm to discover regularities and boundaries within the feature space.

5.3.2 Visualizing the Training Data

Visualizing data is often far more informative than examining tables of numbers. Plotting the training examples allows us to intuitively understand their spatial relationship and overall distribution.



From the plot, we can now clearly see that the two phone categories form distinct clusters within the feature space. Compact phones tend to occupy the lower-right region, while large phones appear in the upper-left region.

This spatial separation strongly suggests that a straight line may be sufficient to act as a decision boundary between the two classes.

5.3.3 Introducing a Linear Decision Boundary

To classify phones, we introduce a straight line of the form:

$$y = Ax$$

Here:

- x represents the phone width
- y represents the phone height

- A controls the slope of the line

We deliberately use the symbols x and y instead of explicit feature names because the line is not acting as a predictor. It does not attempt to predict height from width. Instead, it serves as a **decision boundary** that separates two categories.

For simplicity, we avoid the more general form $y = Ax + B$. Including a bias term would shift the line away from the origin, but it does not add essential insight at this stage.

5.3.4 Starting with a Random Guess

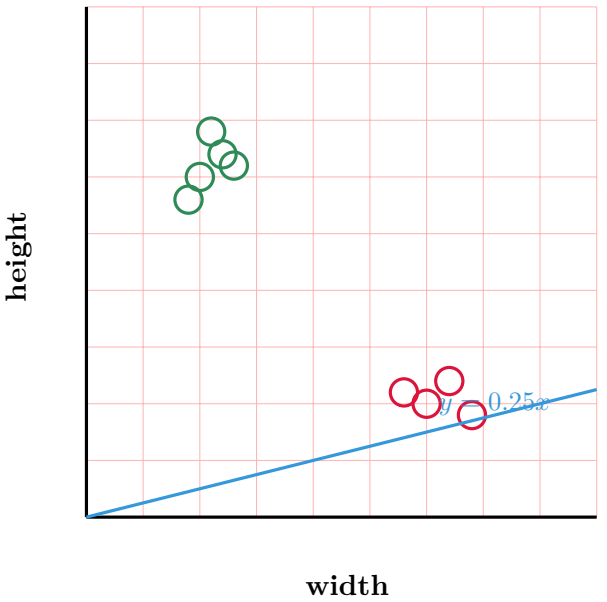
Learning begins with an initial guess. We select a starting value:

$$A = 0.25$$

This gives the initial decision boundary:

$$y = 0.25x$$

We now plot this line together with the expanded training dataset.



Without performing any numerical calculations, we can already see that this initial line is a poor classifier. Most training points lie above the boundary, meaning that the line fails to effectively separate compact phones from large phones.

5.3.5 Learning From Error

To improve the classifier, we must adjust the slope A. The key idea is to use the **error** between the desired behavior and the current behavior to guide this adjustment.

Consider the first training example:

- Width $x = 3.0$
- True height $y = 1.0$

Using the current model:

$$y = 0.25 \times 3.0 = 0.75$$

The model predicts a height of 0.75, but the training data tells us the correct value is 1.0. This difference represents an error.

However, we do not want the line to pass exactly through the training point. Since the line is a classifier rather than a predictor, it should lie slightly above the compact phone, ensuring compact phones remain below the boundary.

We therefore choose a target value:

$$t = 1.1$$

The error E is defined as:

$$E = t - y = 1.1 - 0.75 = 0.35$$

This error tells us not only that the line is incorrect, but also the direction in which it should be adjusted.

Understanding the Role of Error

At this point, we have identified an error for the first training example. The crucial question now is:

> *How should this error be used to refine the slope of the decision boundary?*

To answer this, we must understand the relationship between the slope parameter A and the resulting error in the output of the classifier.

Recall that the decision boundary is defined as:

$$y = Ax$$

For a given input x, the value of y depends directly on A. This means that any error in y must ultimately be caused by an incorrect value of A. If we can determine how much A needs to change to reduce the error, we will have a systematic method for learning.

Relating Error to Parameter Change

Let us denote the correct desired output by t (for target). This is the value of y that we want the classifier to produce for a given input x.

Suppose we adjust the slope A by a small amount. Mathematicians denote a small change using the symbol Δ. Thus, a refined slope can be written as:

$$A + \Delta A$$

With this updated slope, the output of the classifier becomes:

$$t = (A + \Delta A)x$$

Recall that the current output of the classifier is:

$$y = Ax$$

The error E is defined as the difference between the desired target and the actual output:

$$E = t - y$$

Substituting the expressions above:

$$E = (A + \Delta A)x - Ax$$

Expanding and simplifying:

$$E = Ax + (\Delta A)x - Ax$$

$$E = (\Delta A)x$$

This is a remarkably simple and powerful result. It shows that the error is directly proportional to the change in the slope parameter.

Deriving the Update Rule

From the relationship:

$$E = (\Delta A)x$$

we can isolate the required change in the slope:

$$\Delta A = \frac{E}{x}$$

This expression tells us exactly how much the slope A should be adjusted in order to reduce the error for a given training example.

Key Learning Rule

The slope of a linear classifier can be refined using:

$$\Delta A = \frac{E}{x}$$

This update rule directly links classification error to parameter adjustment.

Updating the Slope Using the First Training Example

Let us now apply this update rule to the first training example.

From earlier calculations:

- Error: $E = 0.35$
- Input width: $x = 3.0$

Using the update formula:

$$\Delta A = \frac{0.35}{3.0} = 0.1167$$

The original slope was:

$$A = 0.25$$

After updating:

$$A_{\text{new}} = 0.25 + 0.1167 = 0.3667$$

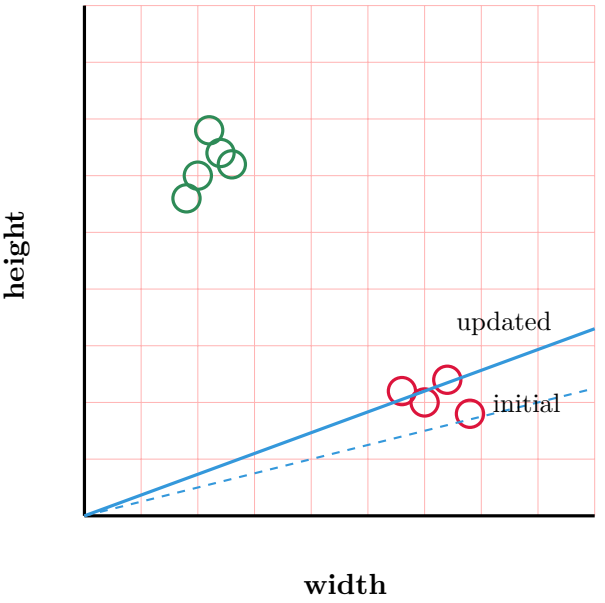
With this updated slope, the classifier now produces:

$$y = 0.3667 \times 3.0 = 1.1$$

which matches the desired target value. The decision boundary has successfully adjusted itself based on the observed error.

Visualizing the Updated Decision Boundary

After processing the first subset of training examples, the decision boundary is adjusted upward. This update reflects the influence of compact phone examples, which lie lower in height but farther along the width axis.



The line has clearly moved upward. This improves its position relative to the cluster of compact phones, pushing more compact examples below the boundary while leaving the taller phones above it.

5.3.6 Learning From the Large-Phone data

We now repeat the same learning procedure using the large-phone training examples. Consider one representative example from this group:

- Width $x = 1.0$
- True height $y = 3.0$

Using the updated slope $A = 0.3667$, the model predicts:

$$y = 0.3667 \times 1.0 = 0.3667$$

This prediction lies far below the actual data point. As before, we select a target value slightly offset from the data point to encourage separation rather than exact prediction. We choose:

$$t = 2.9$$

The resulting error is:

$$E = 2.9 - 0.3667 = 2.5333$$

Applying the same naive update rule:

$$\Delta A = \frac{2.5333}{1.0} = 2.5333$$

The slope is updated to:

$$A_{\text{new}} = 0.3667 + 2.5333 = 2.9$$

This update is extremely large compared to earlier adjustments. It forces the decision boundary to rotate sharply in order to satisfy a single training example.

A Critical Observation

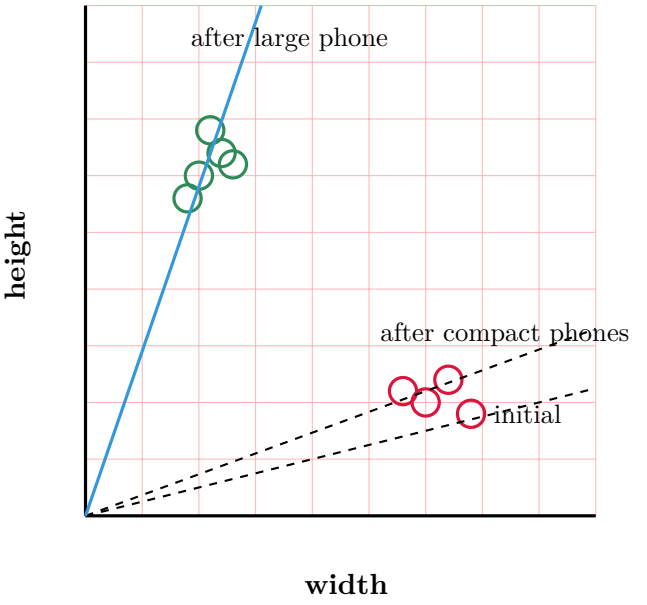
At first glance, this update appears to be successful. The classifier now correctly separates the chosen large-phone example.

However, this apparent success hides a deeper issue. By aggressively updating the slope using a single example, we allow that example to **dominate** the learning process.

In effect, the classifier has drastically reduced the influence of all previously seen compact-phone examples. This represents a lack of *generalization*, where the model becomes too biased toward recent observations.

Visualizing the Problem

The consequences of this aggressive update become clear when we visualize all training examples together with the sequence of decision boundaries.



The final decision boundary now strongly favors the large-phone examples. Several compact phones are incorrectly placed above the line, demonstrating that the classifier has lost its balance.

This behavior is not caused by insufficient data, but by the way updates are applied.

[most]tcolorbox

Critical Concept: Why Naive Updates Fail

When the classifier fully updates its parameters for each individual training example, the final model reflects only the most recent data point.

Earlier examples are effectively overwritten, even if they represent a substantial portion of the dataset. With multiple compact and large phones present, this instability becomes especially problematic.

A reliable classifier must accumulate information from *all* training examples, not oscillate wildly in response to individual ones.

Learning rate: Moderated Learning

To address this instability, we introduce a crucial concept: **moderation**.

Instead of applying the full update ΔA, we apply only a controlled fraction of it. This fraction is governed by a parameter known as the **learning rate** "η".

Moderated updates allow the decision boundary to move gradually, balancing the influence of all training examples. This idea forms the foundation of modern learning algorithms and will be developed in the next section.

5.3.7 Final Parameter Update and Stable Classification

We now introduce a controlled update strategy that allows the classifier to learn gradually from all training examples without overreacting to any single data point.
Instead of applying the full update ΔA, we scale it using a **learning rate** η, where $0 < \eta \ll 1$.

$$A_{\text{new}} = A_{\text{old}} + \eta \times \Delta A$$

The learning rate determines how fast the decision boundary moves. A smaller value ensures stability, while still allowing the model to improve over time.

Aggregating Information From All Training Examples

To prevent dominance by any single phone, we now compute updates using *all* training examples. Each example contributes a small adjustment to the slope of the decision boundary.

For compact phones, the goal is to push the line upward just enough to place them below the boundary. For large phones, the goal is to keep them above the boundary.

After one full pass through the dataset, the slope changes only slightly. Repeating this process over multiple iterations allows the classifier to gradually settle into a balanced position.

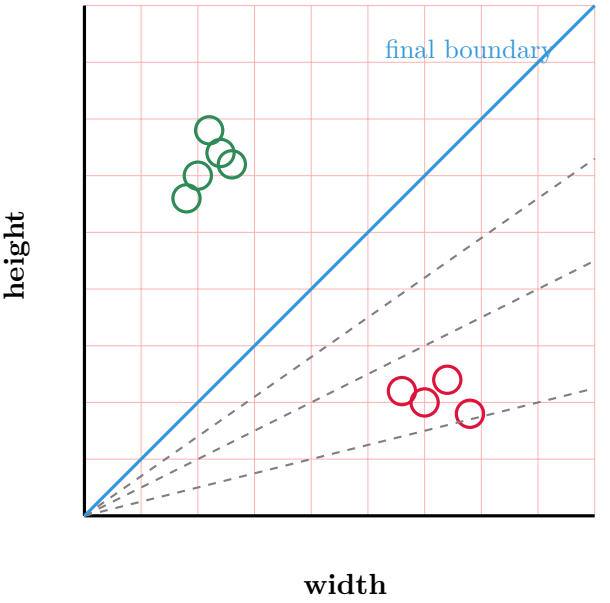
After several learning cycles, the slope converges toward:

$$A \approx 1.0$$

At this value, the line separates the two clusters with minimal conflict.

Visualizing Convergence of the Decision Boundary

The gradual refinement of the decision boundary can be visualized by plotting several intermediate lines along with the final boundary.



The dashed lines represent earlier stages of learning. Each iteration moves the boundary only slightly, preserving information from all training examples. The final line achieves a stable separation between compact and large phones.

5.3.8 Final Classification Rule

With the learned slope $A \approx 1.0$, the classifier uses the rule:

$$\text{If } y > Ax \text{ classify as Large Phone}$$

$$\text{If } y < Ax \text{ classify as Compact Phone}$$

This rule correctly classifies all nine training examples and generalizes naturally to new phones with similar geometric proportions.

Key Takeaways

- Stable learning requires moderated updates
- Learning rate prevents domination by single examples
- Decision boundaries emerge through gradual refinement
- Even simple linear models capture meaningful structure

This completes the learning process. The classifier has transitioned from a random guess to a balanced, data-driven decision boundary = demonstrating the core principle behind all modern machine learning algorithms.

The Limits of Linearity

The simple predictors and classifiers studied so far share a common structure: they take numerical inputs, perform a weighted computation, and produce a single output. As we have seen, such models can be surprisingly effective. However, they are fundamentally limited and cannot solve all problems of practical interest.

Understanding these limitations is essential, because a key design principle of neural networks emerges directly from recognizing where simple linear classifiers fail.

To illustrate this limitation clearly, we now move away from physical examples and consider a class of problems from **Boolean logic**.

5.3.9 Boolean Logic as a Classification Problem

Boolean logic functions operate on binary inputs and produce a binary output. Each input can take the value 0 (false) or 1 (true).

Computers commonly represent:

$$\text{true} = 1, \qquad \text{false} = 0$$

We focus on three fundamental Boolean functions: **AND**, **OR**, and **XOR**.

These functions can be viewed as classification rules that map two inputs (A, B) to a single output.

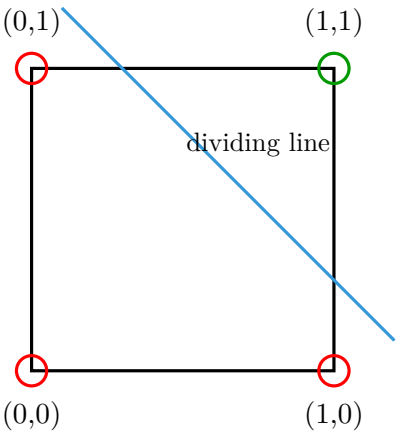
5.3.10 Logical AND

The Boolean AND function is true only when *both* inputs are true.

When plotted in the (A, B) plane, only the point $(1, 1)$ belongs to the positive class. All other points are negative.

Table 5.4: Boolean AND Function

Input A	Input B	AND(A, B)
0	0	0
0	1	0
1	0	0
1	1	**1**



The diagram clearly shows that a single straight line can separate the positive and negative classes. Therefore, the Boolean AND function is **linearly separable**.

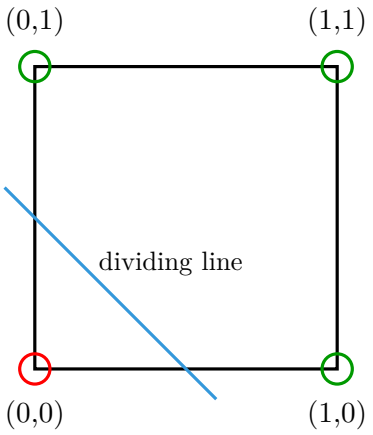
5.3.11 Logical OR

The Boolean OR function is true when *at least one* input is true.

Table 5.5: Boolean OR Function

Input A	Input B	OR(A, B)
0	0	0
0	1	**1**
1	0	**1**
1	1	**1**

Only the point $(0, 0)$ belongs to the negative class.



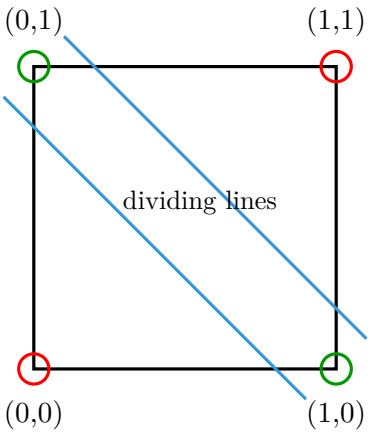
Once again, a straight line can separate the two classes. Thus, the Boolean OR function is also **linearly separable**.

5.3.12 Logical XOR: A Fundamental Limitation

The Boolean XOR (exclusive OR) function is true only when *exactly one* input is true.

Table 5.6: Boolean XOR Function

Input A	Input B	XOR(A, B)
0	0	0
0	1	**1**
1	0	**1**
1	1	0



No single straight line can separate the positive and negative classes. This makes the XOR function **not linearly separable**.

Key Insight: The Birth of Neural Networks

A simple linear classifier fails whenever the underlying problem cannot be separated by a single straight line.

The **XOR problem** demonstrates this limitation in the clearest possible way.

The solution is not to abandon linear classifiers, but to **combine multiple linear decision boundaries**. This idea lies at the very heart of neural networks, where many simple units work together to solve complex, non-linear problems.

5.4 Neurons: Nature's Computing Units

For many years, scientists were puzzled by an apparent contradiction. Biological brains, even very small ones, perform remarkably complex tasks, yet they operate far more slowly and contain far fewer computing elements than modern digital computers.

For example, a pigeon's brain or a fruit fly's nervous system far outperforms traditional computers at tasks such as navigation, pattern recognition, and learning from experience, despite running at slow biological speeds and using noisy, imprecise signals.

The key difference does not lie in raw speed or storage capacity, but in **architecture**. Conventional computers process information sequentially, using exact and deterministic operations. Biological brains, in contrast,

process information in a massively parallel and distributed fashion, where fuzziness and approximation are essential features, not flaws.

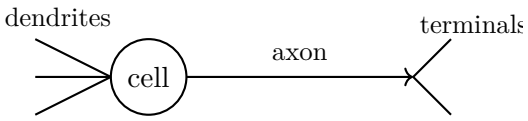
To understand this difference, we begin with the fundamental building block of biological brains: the neuron.

5.4.1 The Biological Neuron

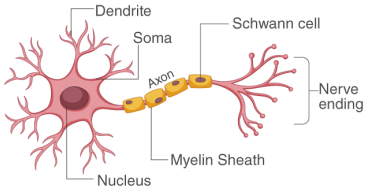
A biological neuron is a specialized cell that transmits electrical signals. Although neurons come in many forms, they share a common structure.

- **Dendrites** receive incoming signals
- The signal travels through the cell body
- If strong enough, it propagates along the **axon**
- The signal reaches the **terminals** and is passed onward

This process allows sensory signals such as light, sound, pressure, and temperature to be transmitted through the nervous system and processed by the brain.



(a) Artificial neuron model



(b) Biological neuron (illustration)

Figure 5.14: Comparison between an artificial neuron and a biological neuron

Even organisms with very small numbers of neurons can perform surprisingly sophisticated behaviors. A fruit fly has roughly 10^5 neurons, while a simple nematode worm has only 302 neurons, yet it can move, sense its environment, and respond adaptively.

This suggests that intelligence does not arise solely from the number of components, but from how they are connected and interact.

5.4.2 Neurons as Input–Output Devices

At a high level, a neuron behaves like a machine: it takes an input signal and produces an output signal. This strongly resembles the predictive and classification models we have already studied.

However, biological neurons do not behave like simple linear functions. Their output is not merely a weighted version of the input.

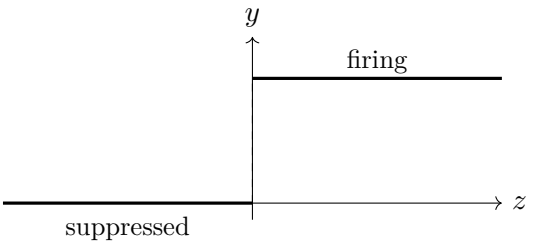
Instead, neurons exhibit a **threshold effect**. Weak signals are suppressed, while sufficiently strong signals trigger a response. Neuroscientists describe this behavior by saying that neurons *fire* when the input exceeds a threshold.

An intuitive analogy is water in a cup: nothing happens until the cup is filled to the brim, at which point water spills over.

This behavior is captured mathematically by what is known as an **activation function**.

5.4.3 Threshold and Activation Functions

The simplest activation function is a step function. Below a certain input level, the output is zero. Once the threshold is crossed, the output jumps to a fixed value.

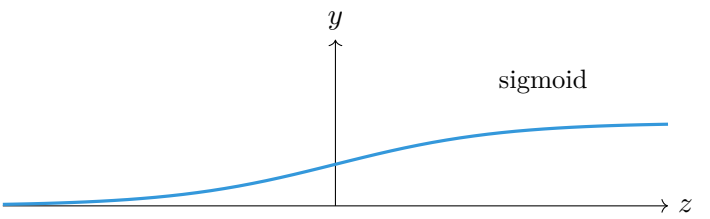


While effective, the step function is abrupt and biologically unrealistic. Nature rarely operates with sharp discontinuities.

A smoother and more realistic alternative is the **sigmoid function**, which produces an S-shaped curve.

$$y = \frac{1}{1 + e^{-z}}$$

For small input values, the output is close to zero. As the input increases, the output rises smoothly toward one. When $z = 0$, the output equals $\frac{1}{2}$.



The sigmoid function is especially useful because it is smooth, differentiable, and easy to work with mathematically. For these reasons, it is widely used in artificial neural networks.

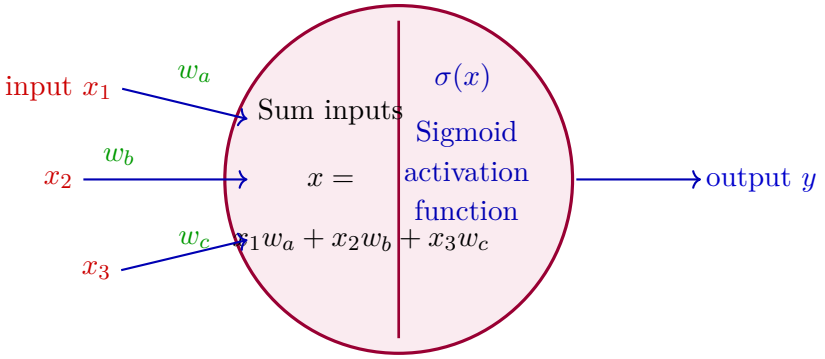


Figure 5.15: Single neuron with weighted inputs and sigmoid activation function

5.4.4 Signal Processing in a Single Neuron

Figure 5.15 illustrates the internal working of a single artificial neuron. Each neuron receives multiple input signals $\{a, b, c\}$, where every input represents a feature or measurement from the data.

Each input is multiplied by an associated weight $\{w_a, w_b, w_c\}$. These weights control the importance of each input in the decision-making process. Inputs with larger weights contribute more strongly to the neuron's output.

The neuron then computes a weighted sum of all inputs, given by

$$x = x_1w_a + x_2w_b + x_3w_c.$$

This summation stage combines the incoming information into a single scalar value.

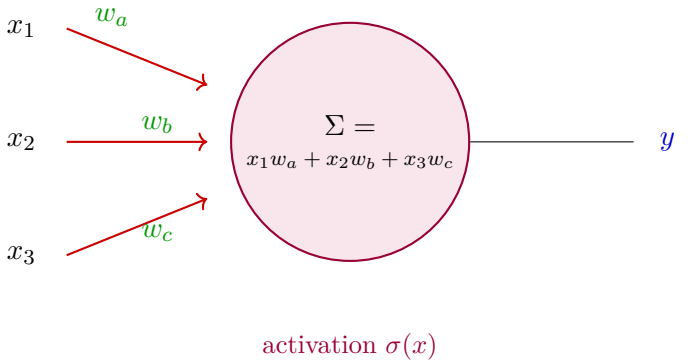
Next, the combined signal x is passed through a sigmoid activation function, denoted as $\sigma(x)$. The activation function introduces non-linearity into the model and squashes the output into a limited range, typically between 0 and 1.

Finally, the neuron produces the output y, which can be interpreted as:

- a probability,
- a decision signal, or
- an intermediate value passed to the next layer.

This simple processing unit forms the fundamental building block of neural networks, and by combining many such neurons into layers, complex patterns and relationships in data can be learned.

In an artificial neuron, this combination is modeled by summing the inputs, which are then passed through an activation function.



5.4.5 Sigmoid Function as an Activation (Firing) Function

The sigmoid function is a commonly used activation (firing) function in artificial neural networks. It determines the activation level of a neuron by smoothly mapping the weighted sum of its inputs to a bounded output.

Mathematically, the sigmoid function is defined as

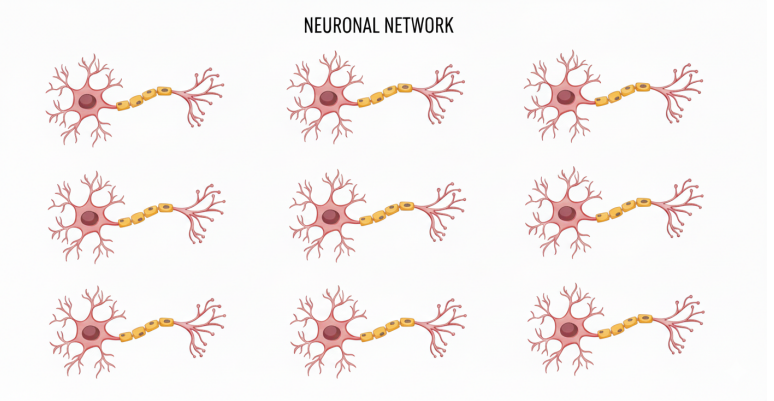
$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Here, x represents the weighted sum of input signals. The sigmoid function transforms this input into an output value in the range $[0, 1]$, which can be interpreted as the neuron's activation strength or the probability of firing.

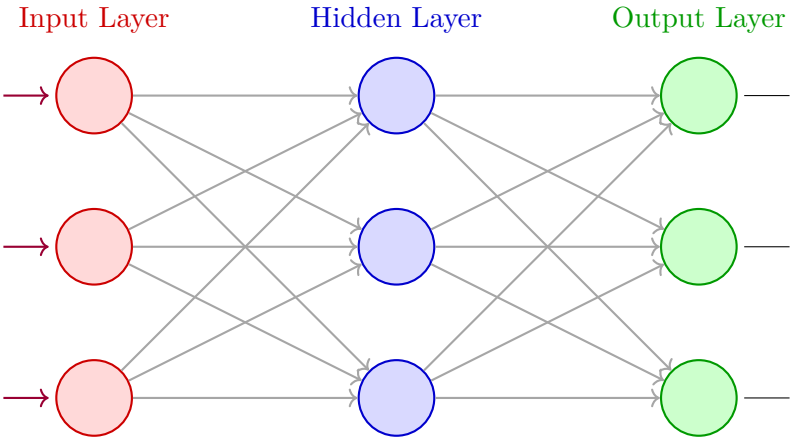
For large positive values of x, the output approaches 1, indicating strong activation. For large negative values, the output approaches 0, and the neuron remains inactive. Near $x = 0$, the sigmoid function provides a smooth transition between these states, which supports stable learning and gradual weight updates.

5.5 Networks of Neurons

Neurons do not operate in isolation. Each neuron receives signals from many others and transmits signals onward.



This connectivity inspires the structure of artificial neural networks.

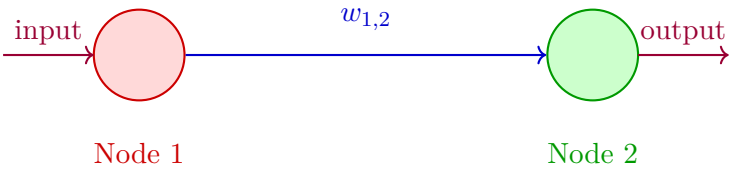


Each neuron connects to many others, creating a layered structure. This full connectivity is not always minimal, but it simplifies implementation and allows learning to decide which connections matter.

Learning Through Weights

Learning occurs by adjusting the **strength of connections** between neurons. These strengths are called **weights**.

A small weight weakens a signal. A large weight amplifies it. A zero weight effectively removes a connection.



During learning, some weights grow stronger, while others shrink toward zero. Connections that are not useful are automatically de-emphasized.

This adaptive refinement of weights allows neural networks to learn complex patterns from data.

Key Points: The Biological to Artificial Bridge

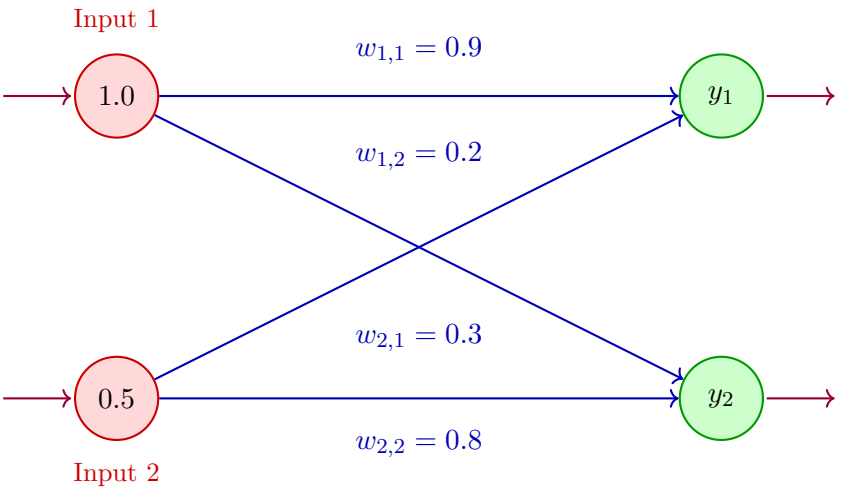
- **Biological Efficiency:** Brains perform sophisticated tasks despite slow operation and noisy signals.
- **Parallel Processing:** Neurons process information in parallel using thresholded, non-linear responses.
- **Modeling Behavior:** Artificial neurons model this behavior using **activation functions** and weighted inputs.
- **Learning Mechanism:** Neural networks learn by adjusting the strengths (weights) of connections between neurons.

5.5.1 Following Signals Through a Neural Network

Earlier, we introduced a neural network composed of multiple layers of interconnected neurons. While such diagrams may look visually impressive, they can also appear intimidating when we think about how signals actually flow through the network and how outputs are computed.

Although the calculations involved are indeed laborious, it is important to work through a simple example by hand. Doing so ensures we clearly understand what is happening inside a neural network, even if we later rely on computers to perform the calculations automatically.

To keep things manageable, we will analyze a very small neural network consisting of only two layers, each containing two neurons.



Inputs and Weights

Let the two input values be:

$$x_1 = 1.0, \qquad x_2 = 0.5$$

Each connection between neurons has an associated weight. We initialize these weights with random values:

$$w_{1,1} = 0.9, \quad w_{1,2} = 0.2, \quad w_{2,1} = 0.3, \quad w_{2,2} = 0.8$$

Random initialization is not arbitrary; it allows the learning process to gradually refine these values as training progresses, just as we previously adjusted the slope in simple linear classifiers.

The Input Layer

The first layer is called the *input layer*. It performs no computation and does not apply an activation function. Its sole purpose is to pass the input values forward into the network. The output at layer 2 are computed as follows:

Step-by-Step Computation: Node 1 (Layer 2)

First, we compute the combined input (z_1) as a **weighted sum** of the signals arriving from the previous layer:

$$z_1 = (1.0 \times 0.9) + (0.5 \times 0.3) = 1.05$$

Next, this value is transformed by the **sigmoid activation function** to determine the final output of the node:

$$y_1 = \sigma(1.05) = \frac{1}{1 + e^{-1.05}}$$

Result: $y_1 \approx \mathbf{0.7408}$

Step-by-Step Computation: Node 2 (Layer 2)

We repeat the process for the second neuron. First, calculate the **weighted sum** (z_2) of the inputs:

$$z_2 = (1.0 \times 0.2) + (0.5 \times 0.8) = 0.6$$

Next, apply the **sigmoid activation function** to obtain the final output (y_2):

$$y_2 = \sigma(0.6) = \frac{1}{1 + e^{-0.6}}$$

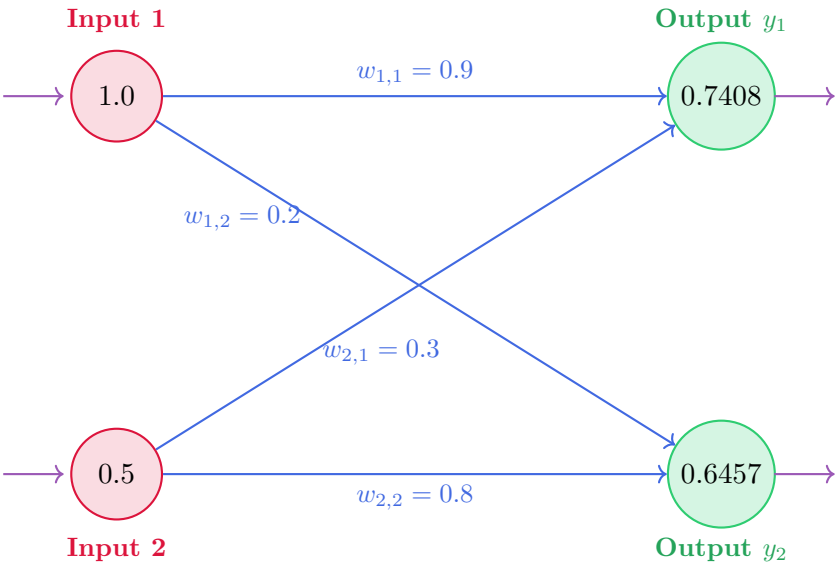
Result: $y_2 \approx \mathbf{0.6457}$

Final Output Vector: Layer 2 Result

After completing the activation of all neurons in the output layer, the neural network produces the following final prediction vector:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 0.7408 \\ 0.6457 \end{bmatrix}$$

This vector represents the network's confidence scores for each output class.



Need a Better Mathematical Tool

Even for this extremely small network, computing the outputs required multiple steps and careful bookkeeping. Performing such calculations by hand for networks with many layers and hundreds of neurons would be impractical and error-prone.

Fortunately, mathematics provides a concise and elegant way to express these computations using *matrices*. Matrix-based formulations allow both humans and computers to represent neural networks compactly and evaluate them efficiently.

5.5.2 Matrix Multiplication: Step-by-Step Explanation

Matrix multiplication is a core mathematical operation used throughout machine learning and neural networks. It provides a compact and efficient way to compute multiple weighted sums at the same time, which is exactly what happens when signals pass through a layer of neurons.

Multiplication of Two 3×3 Matrices

Consider two matrices:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, \qquad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}.$$

The product

$$\mathbf{C} = \mathbf{AB}$$

results in another 3×3 matrix, where each element of $\mathbf{C}$ is obtained by taking the dot product of a row of $\mathbf{A}$ with a column of $\mathbf{B}$.

For example:

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31},$$

$$c_{12} = a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32},$$

$$c_{13} = a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33}.$$

Repeating this process for all rows and columns gives:

$$\mathbf{C} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}.$$

Thus, each element of the result is a weighted sum of three values, reflecting how information is combined across dimensions.

To better understand matrix multiplication, we now illustrate the process using simple Examples. These examples demonstrate how weighted sums are computed explicitly, which directly mirrors the calculations performed inside neural networks.

Example: Multiplication of Two 3×3 Matrices

Consider the matrices:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \qquad \mathbf{B} = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}.$$

The product

$$\mathbf{C} = \mathbf{AB}$$

is computed by taking the dot product of each row of $\mathbf{A}$ with each column of $\mathbf{B}$.

$$c_{11} = (1)(1) + (2)(0) + (3)(1) = 4$$

$$c_{12} = (1)(0) + (2)(1) + (3)(1) = 5$$

$$c_{13} = (1)(2) + (2)(1) + (3)(0) = 4$$

$$c_{21} = (4)(1) + (5)(0) + (6)(1) = 10$$

$$c_{22} = (4)(0) + (5)(1) + (6)(1) = 11$$

$$c_{23} = (4)(2) + (5)(1) + (6)(0) = 13$$

$$c_{31} = (7)(1) + (8)(0) + (9)(1) = 16$$
$$c_{32} = (7)(0) + (8)(1) + (9)(1) = 17$$
$$c_{33} = (7)(2) + (8)(1) + (9)(0) = 22$$

Thus, the resulting matrix is:

$$\mathbf{C} = \begin{bmatrix} 4 & 5 & 4 \\ 10 & 11 & 13 \\ 16 & 17 & 22 \end{bmatrix}.$$

Multiplication of a 3×3 Matrix with a 3×1 Vector

Now consider a matrix–vector multiplication, which commonly appears in neural networks when computing the output of a layer.

Let

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

The product

$$\mathbf{y} = \mathbf{W}\mathbf{x}$$

produces a 3×1 output vector:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}.$$

Each output element is computed as:

$$y_1 = w_{11}x_1 + w_{12}x_2 + w_{13}x_3,$$
$$y_2 = w_{21}x_1 + w_{22}x_2 + w_{23}x_3,$$
$$y_3 = w_{31}x_1 + w_{32}x_2 + w_{33}x_3.$$

Each row of the matrix acts like a neuron, combining inputs using weighted sums. The resulting vector is then typically passed through an activation function to introduce non-linearity.

In summary, matrix multiplication provides a powerful and structured way to model how information flows and transforms inside neural networks.

Now consider a matrix–vector multiplication, which frequently occurs when computing the output of a neural network layer.

Let

$$\mathbf{W} = \begin{bmatrix} 0.2 & 0.5 & 0.3 \\ 0.6 & 0.1 & 0.4 \\ 0.7 & 0.2 & 0.8 \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} 1.0 \\ 0.5 \\ 0.2 \end{bmatrix}.$$

The output vector is given by:

$$\mathbf{y} = \mathbf{W}\mathbf{x}.$$

Each element is computed as follows:

$$y_1 = (0.2)(1.0) + (0.5)(0.5) + (0.3)(0.2) = 0.51$$
$$y_2 = (0.6)(1.0) + (0.1)(0.5) + (0.4)(0.2) = 0.73$$
$$y_3 = (0.7)(1.0) + (0.2)(0.5) + (0.8)(0.2) = 0.96$$

Thus, the final output vector is:

$$\mathbf{y} = \begin{bmatrix} 0.51 \\ 0.73 \\ 0.96 \end{bmatrix}.$$

These Examples clearly show how matrix multiplication performs multiple weighted sums in a structured and efficient manner, forming the mathematical backbone of neural network computations.

In the next section, we introduce matrix notation and show how it simplifies neural network calculations dramatically.

5.5.3 Matrix Multiplication in a 2×2 Neural Network Layer

In neural networks, the flow of information from one layer to the next is most naturally expressed using matrix multiplication. This mathematical operation allows multiple neurons to compute their outputs *simultaneously* by combining inputs with their corresponding weights.

We consider here a simple yet instructive case: a neural network layer with two inputs and two neurons, as illustrated in Figure 5.16.

Weight Matrix and Input Vector

Let the weight matrix be

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{bmatrix},$$

where each weight $w_{i,j}$ represents the strength of the connection from input i to neuron j.

The input vector is defined as

$$\mathbf{x} = \begin{bmatrix} input_1 \\ input_2 \end{bmatrix}.$$

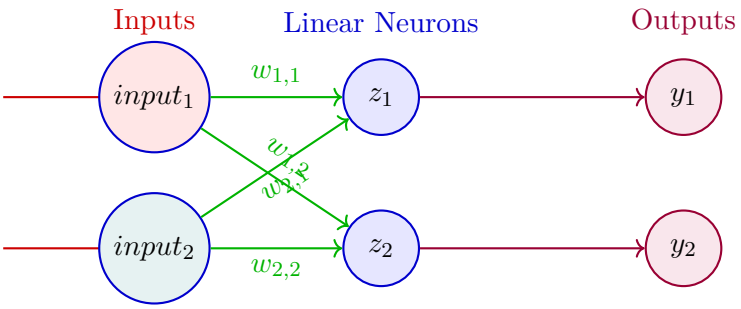


Figure 5.16: A 2×2 neural network layer with explicit input and output arrows

Layer Computation via Matrix Multiplication

The linear computation performed by the neural network layer is

$$\mathbf{z} = \mathbf{W}\mathbf{x},$$

which results in the output (pre-activation) vector

$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}.$$

Each neuron computes a weighted sum of all inputs:

$$z_1 = (input_1 \cdot w_{1,1}) + (input_2 \cdot w_{2,1}),$$

$$z_2 = (input_1 \cdot w_{1,2}) + (input_2 \cdot w_{2,2}).$$

These values are then passed through an activation function to produce the final neuron outputs.

Neural Network Interpretation

- Each row of the weight matrix corresponds to one neuron.
- Each neuron receives *all inputs* and combines them using learned weights.
- Matrix multiplication enables the parallel computation of neuron activations.

5.5.4 Weight Update in a Three-Layer Neural Network

We now extend the learning process to a more realistic neural network architecture consisting of **three layers**:

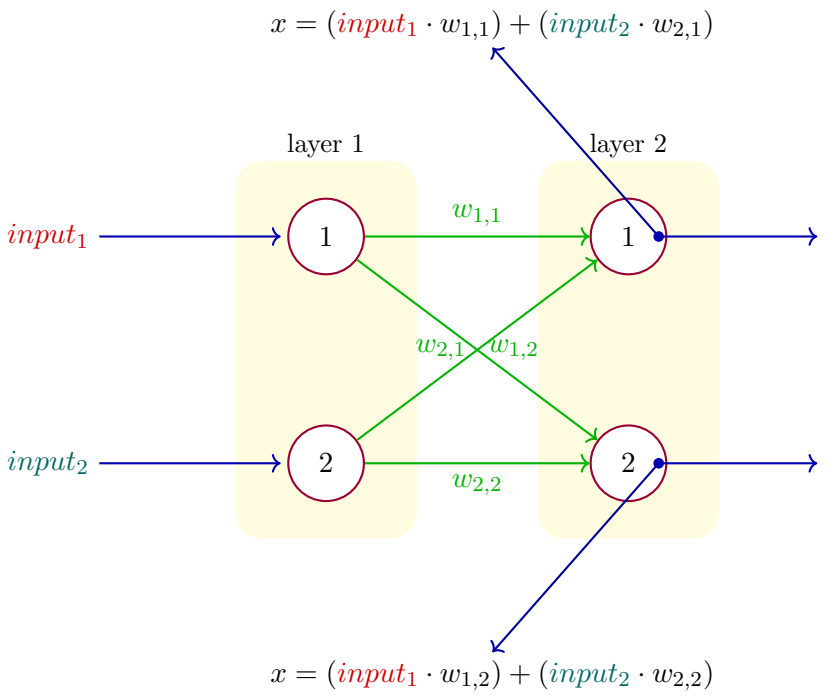


Figure 5.17: Matrix–vector multiplication in a 2×2 neural network layer

- an **input layer**,
- a **hidden layer**,
- and an **output layer**.

Each layer contains three nodes, forming a 3×3 network structure. This architecture allows the network to model more complex, non-linear relationships than a simple two-layer network.

Network Structure and Notation

Let:

- Inputs be x_1, x_2, x_3
- Hidden-layer outputs be h_1, h_2, h_3
- Final outputs be y_1, y_2, y_3

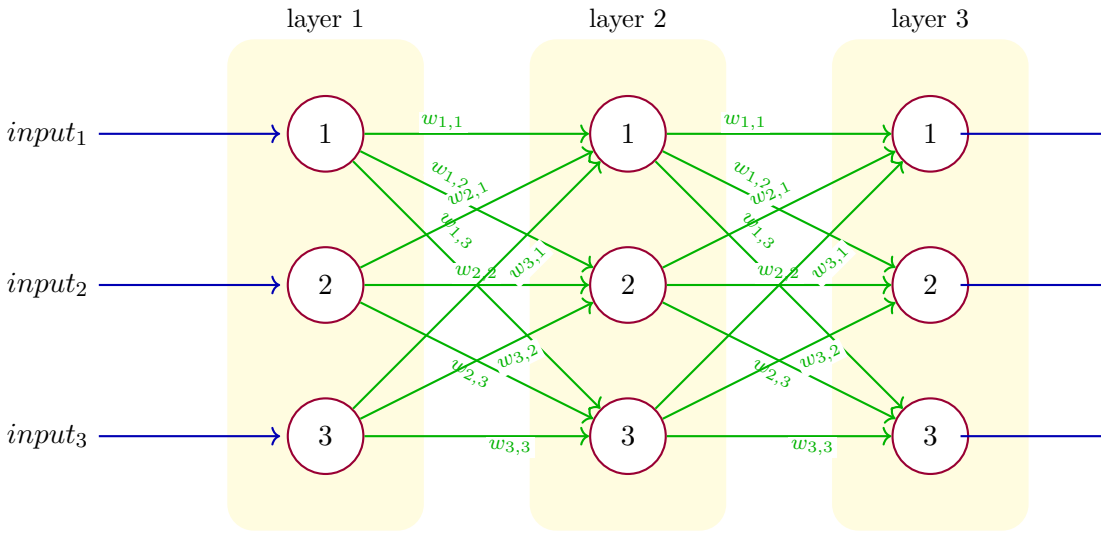


Figure 5.18: Fully connected three-layer neural network with non-overlapping weight labels

We define two weight matrices:

$$\mathbf{W}^{(1)} = \begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} & w_{1,3}^{(1)} \\ w_{2,1}^{(1)} & w_{2,2}^{(1)} & w_{2,3}^{(1)} \\ w_{3,1}^{(1)} & w_{3,2}^{(1)} & w_{3,3}^{(1)} \end{bmatrix}$$

connecting the input layer to the hidden layer, and

$$\mathbf{W}^{(2)} = \begin{bmatrix} w_{1,1}^{(2)} & w_{1,2}^{(2)} & w_{1,3}^{(2)} \\ w_{2,1}^{(2)} & w_{2,2}^{(2)} & w_{2,3}^{(2)} \\ w_{3,1}^{(2)} & w_{3,2}^{(2)} & w_{3,3}^{(2)} \end{bmatrix}$$

connecting the hidden layer to the output layer.

Forward Pass (Summary)

The hidden-layer inputs are computed as:

$$z_j^{(1)} = \sum_{i=1}^{3} w_{i,j}^{(1)} x_i$$

and the hidden-layer outputs using the sigmoid activation function:

$$h_j = \sigma(z_j^{(1)})$$

Similarly, the output-layer inputs are:

$$z_k^{(2)} = \sum_{j=1}^{3} w_{j,k}^{(2)} h_j$$

and the final outputs are:

$$y_k = \sigma(z_k^{(2)})$$

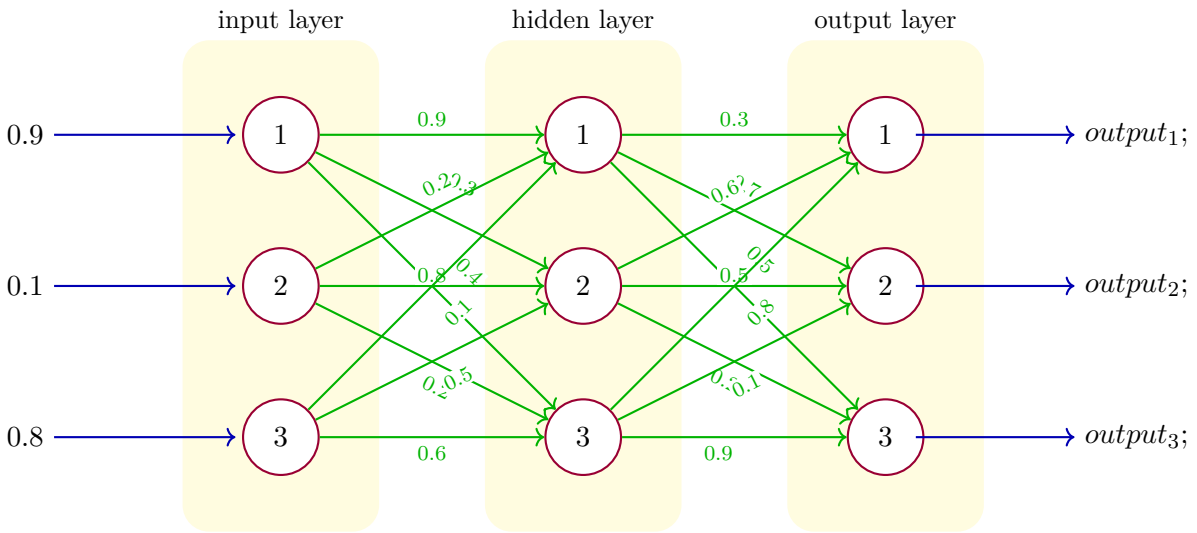


Figure 5.19: Three-layer neural network illustrating matrix-based weight multiplication with non-overlapping labels

Example: Signal Flow Through the Neural Network

Figure 5.19 illustrates a fully connected three-layer neural network consisting of an input layer, a hidden layer, and an output layer. Each connection between neurons is associated with a numerical weight, as explicitly labeled in the diagram.

The input vector is given by

$$\mathbf{I} = \begin{bmatrix} 0.9 \\ 0.1 \\ 0.8 \end{bmatrix},$$

where each value represents the signal applied to one input neuron.

Step 1: Weighted Summation at the Hidden Layer Each hidden neuron receives signals from all input neurons. These signals are multiplied by their corresponding weights and then summed.

Hidden neuron 1:

$$z_1^{(h)} = (0.9 \times 0.9) + (0.1 \times 0.2) + (0.8 \times 0.1) = 0.81 + 0.02 + 0.08 = 0.91$$

Hidden neuron 2:

$$z_2^{(h)} = (0.9 \times 0.3) + (0.1 \times 0.8) + (0.8 \times 0.5) = 0.27 + 0.08 + 0.40 = 0.75$$

Hidden neuron 3:

$$z_3^{(h)} = (0.9 \times 0.4) + (0.1 \times 0.2) + (0.8 \times 0.6) = 0.36 + 0.02 + 0.48 = 0.86$$

Step 2: Activation Using the Sigmoid Function Each summed signal is passed through the sigmoid activation function

$$\sigma(x) = \frac{1}{1 + e^{-x}},$$

which squashes the input into the range $(0, 1)$.

$$h_1 = \sigma(0.91) \approx 0.713, \qquad h_2 = \sigma(0.75) \approx 0.679, \qquad h_3 = \sigma(0.86) \approx 0.703$$

Thus, the hidden layer output vector becomes

$$\mathbf{H} = \begin{bmatrix} 0.713 \\ 0.679 \\ 0.703 \end{bmatrix}.$$

Step 3: Weighted Summation at the Output Layer The outputs from the hidden layer are again multiplied by the weights connecting to the output layer.

Output neuron 1:

$$z_1^{(o)} = (0.713{\times}0.3){+}(0.679{\times}0.6){+}(0.703{\times}0.8) = 0.214{+}0.407{+}0.562 = 1.183$$

Output neuron 2:

$$z_2^{(o)} = (0.713{\times}0.7){+}(0.679{\times}0.5){+}(0.703{\times}0.1) = 0.499{+}0.340{+}0.070 = 0.909$$

Output neuron 3:

$$z_3^{(o)} = (0.713{\times}0.5){+}(0.679{\times}0.2){+}(0.703{\times}0.9) = 0.357{+}0.136{+}0.633 = 1.126$$

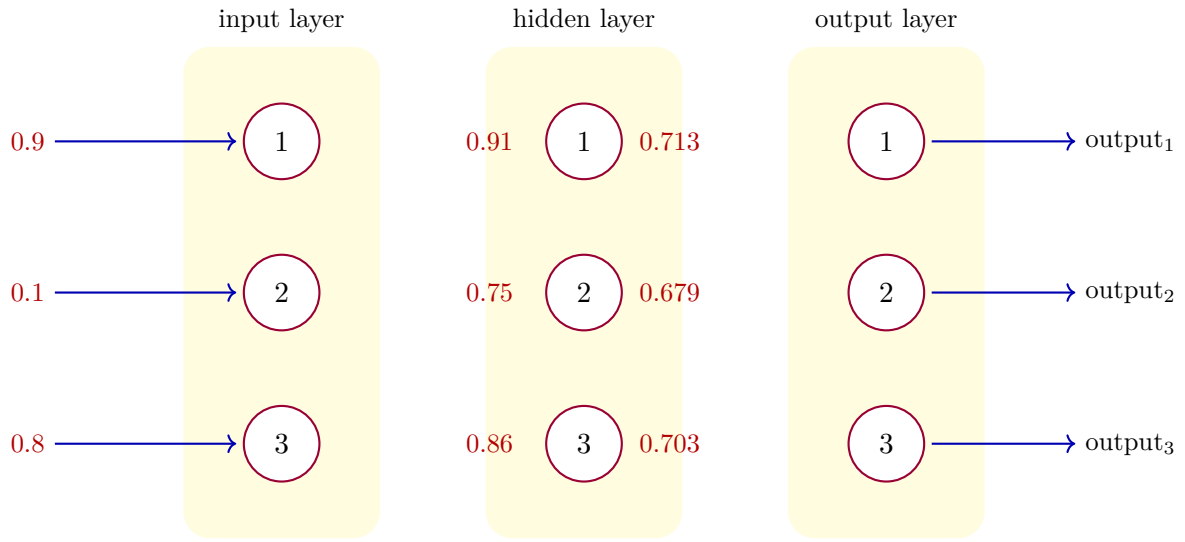


Figure 5.20: Neural network showing only node input and output values at hidden and output layers

Step 4: Final Output via Sigmoid Activation Applying the sigmoid function once more yields the final network outputs:

$$\text{output}_1 = \sigma(1.183) \approx 0.765$$
$$\text{output}_2 = \sigma(0.909) \approx 0.713$$
$$\text{output}_3 = \sigma(1.126) \approx 0.755$$

Hence, the final output vector of the neural network is

$$\mathbf{O} = \begin{bmatrix} 0.765 \\ 0.713 \\ 0.755 \end{bmatrix}.$$

Interpretation This detailed signal flow demonstrates how a neural network transforms raw input signals into meaningful outputs through:

- weighted summation,
- nonlinear activation using the sigmoid function,
- and layered matrix-based computation.

The diagram visually mirrors these calculations, making the correspondence between weights, signals, and outputs explicit and intuitive.

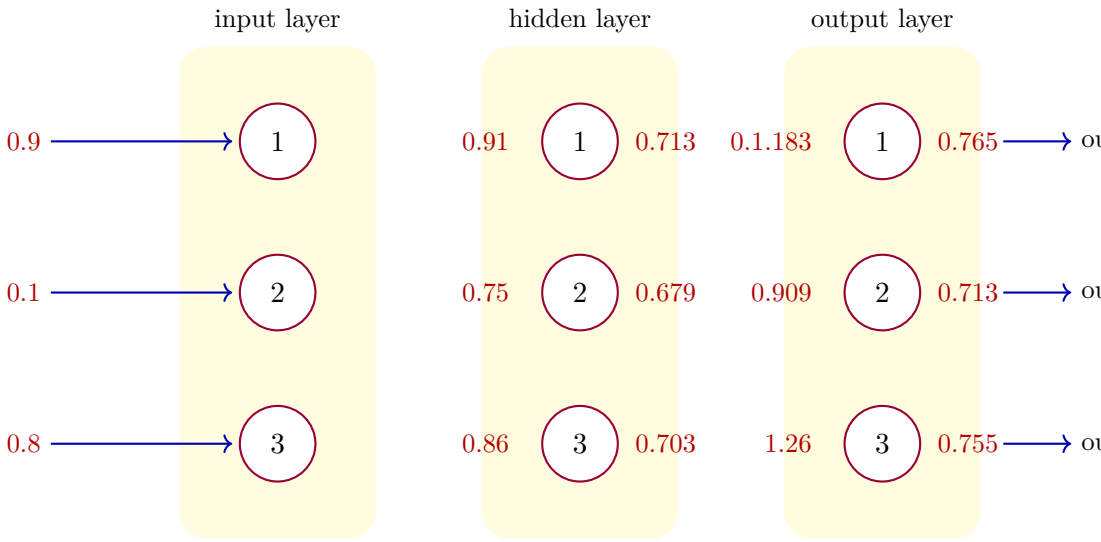


Figure 5.21: Neural network showing only node input and output values at hidden and output layers

Learning Weights From More Than One Node

Previously, we refined a simple linear classifier by adjusting a single weight connected to an output node. In that case, learning was straightforward because only one node contributed to the output and its error.

However, real neural networks are more complex. In practice, an output node usually receives signals from *multiple* preceding nodes. This raises an important question:

How should the output error be used to update more than one weight?

Figure 5.22 illustrates this situation. Two input nodes feed signals into a single output node through links with different weights.

It does not make sense to apply the *entire* output error to only one of these weights, because both links contributed to the final output value. The error exists precisely because *multiple connections* influenced the output.

Although it is theoretically possible that only one weight caused the error, this chance is extremely small. Even if a weight that was already correct is adjusted slightly in the wrong direction, later learning iterations will correct it.

Equal Error Splitting A simple idea is to divide the output error equally among all contributing connections. If two nodes feed into an output node, each weight receives one-half of the error.

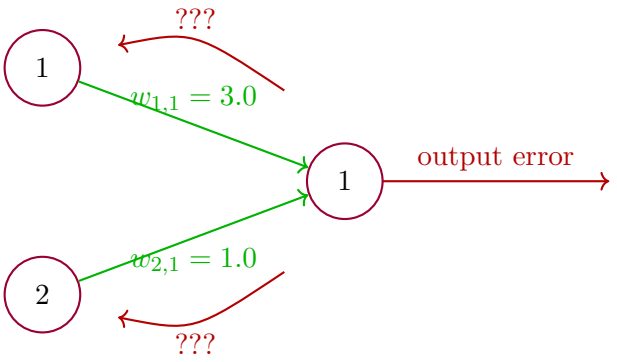


Figure 5.22: An output error caused by multiple contributing nodes

This approach is shown in Figure 5.23. While simple, it ignores the fact that some connections contribute more strongly than others.

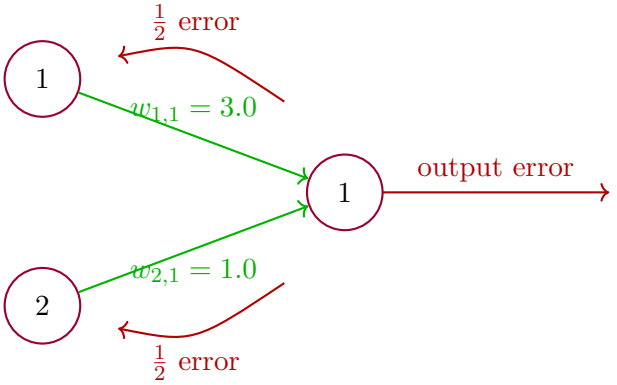


Figure 5.23: Equal splitting of output error

Weighted Error Splitting A more intuitive and effective idea is to divide the error *proportionally* according to the strength of each connection.

In Figure 5.24, the two weights are

$$w_{1,1} = 3.0, \qquad w_{2,1} = 1.0.$$

The total contribution is $3.0 + 1.0 = 4.0$. Therefore:

$$\text{Error share for } w_{1,1} = \frac{3}{4}\,\text{error}, \qquad \text{Error share for } w_{2,1} = \frac{1}{4}\,\text{error}.$$

The larger weight receives a larger portion of the error because it contributed more strongly to the output.

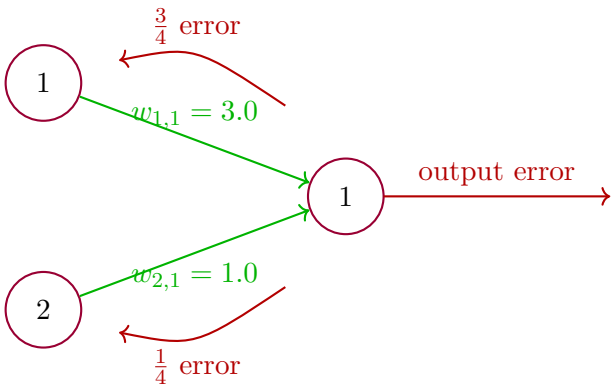


Figure 5.24: Error distributed proportionally to connection weights

This principle naturally extends to many more nodes. If an output node had 100 incoming connections, the error would be split across all 100 weights in proportion to their sizes.

We now see that weights serve two important roles:

- They propagate signals forward through the network.
- They propagate error backward during learning.

This backward flow of error is why the learning process is called **back-propagation**.

If the output layer contains multiple output nodes, each output node produces its own error, which is independently distributed backward across its incoming connections in the same way.

5.5.5 Backpropagating Errors From More Output Nodes

Consider a simple neural network with two input nodes and two output nodes, as illustrated in Figure **??**. When a network has not yet been trained, it is extremely likely that *both* output nodes will produce errors. Each of these errors must be used to refine the internal link weights of the network.

Let the inputs be denoted by i_1 and i_2, and the outputs by o_1 and o_2. The corresponding target values from the training data are t_1 and t_2. The output errors are defined as

$$e_1 = t_1 - o_1, \qquad e_2 = t_2 - o_2.$$

Each output node receives signals from the input layer through weighted links. For output node 1, these links have weights w_{11} and w_{21}. For output node 2, the links have weights w_{12} and w_{22}.

Key idea: The error at an output node is split across its incoming links in proportion to their weights. This ensures that links which contributed more strongly to the output are adjusted more strongly.

Splitting the Error at the First Output Node

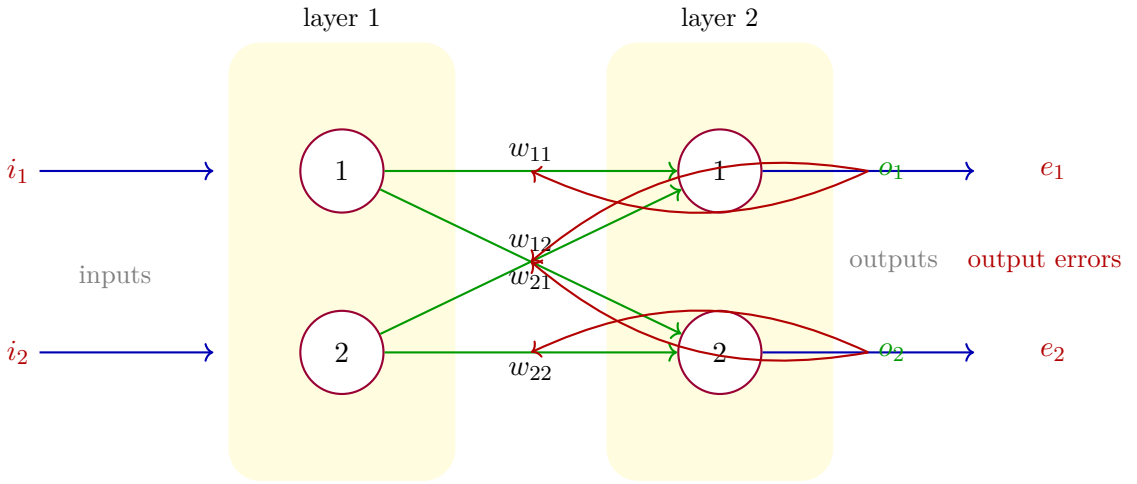


Figure 5.25: Backpropagation of output errors along individual weighted connections

The error e_1 must be used to refine both w_{11} and w_{21}. The fraction of e_1 used to update w_{11} is

$$\frac{w_{11}}{w_{11} + w_{21}},$$

and similarly, the fraction of e_1 used to update w_{21} is

$$\frac{w_{21}}{w_{11} + w_{21}}.$$

These expressions simply encode the idea that a larger weight receives a larger share of the error.

Example: If $w_{11} = 6$ and $w_{21} = 3$, then

$$\frac{6}{6+3} = \frac{2}{3}, \qquad \frac{3}{6+3} = \frac{1}{3}.$$

Thus, two-thirds of the error e_1 is used to refine w_{11} and one-third is used to refine w_{21}.

If the weights are equal, for example $w_{11} = w_{21} = 4$, then

$$\frac{4}{4+4} = \frac{1}{2},$$

and the error is split equally, as expected.

Splitting the Error at the Second Output Node

Exactly the same reasoning applies to the second output node. The error

$$e_2 = t_2 - o_2$$

is split across the links with weights w_{12} and w_{22} in proportion to their magnitudes. There is no dependence between the links feeding into different output nodes, so the refinement process for one output node does not interfere with the other.

Why this is Called Backpropagation

We now see that the weights play two distinct roles:

- They propagate signals *forward* from the input layer to the output layer.
- They propagate errors *backward* from the output layer into the network.

This backward flow of error information is why the learning method is called *backpropagation*.

The next natural question is what happens when the network has more than two layers. How do we update weights in layers further back from the output layer? This is exactly what we will explore next.

5.5.6 Weight Update Using the backpropagation of Error at the Output Layer

Output Layer Error and Weight Update Mechanism

In supervised learning, the adjustment of network weights is guided by the difference between the desired output and the actual output produced by the network. This difference is quantified in the form of an *error signal*, which drives the learning process.

Let the target outputs corresponding to the output neurons be denoted by t_1, t_2, t_3, and let the actual outputs produced by the network be y_1, y_2, y_3. The error at the k-th output neuron is defined as:

$$e_k = t_k - y_k$$

This error represents the discrepancy between the desired response and the network's prediction. A positive error indicates that the neuron output is smaller than desired, while a negative error implies an excessive response.

Output Layer Error and Weight Update Mechanism

Computation of the Delta Term

To update the weights connected to the output layer, the error must be combined with the sensitivity of the activation function. For neurons using the sigmoid activation function, the *delta term* at the output layer is given by:

$$\delta_k^{(2)} = e_k \, y_k(1 - y_k)$$

Here:

- e_k propagates the magnitude and direction of the output error,
- $y_k(1 - y_k)$ is the derivative of the sigmoid activation function, which measures how responsive the neuron output is to changes in its input.

The delta term therefore captures both the error information and the local slope of the activation function, ensuring that weight updates are scaled appropriately.

Role in Weight Adjustment

The computed delta term determines how strongly each output neuron should adjust its incoming weights. If the error is large and the neuron operates in the sensitive (non-saturated) region of the sigmoid function, the weight updates will be significant. Conversely, when the neuron is saturated (output close to 0 or 1), the derivative becomes small, resulting in smaller weight updates.

This mechanism ensures stable learning by preventing excessively large weight changes while still allowing the network to reduce output errors effectively.

v

Backpropagating Errors to Multiple Layers

Step 1: Calculating the Output Error (e_{output})

Learning begins at the **Output Layer**. This is the only point where we have a known "Target" to compare against our prediction.

The difference between the target (t) and the actual output (y) is the **output error**:

$$e_{\text{output}} = \text{Target} - \text{Actual Output}$$

These error signals represent the "mistake" each output neuron made. We use these values to refine the weights (w_{ho}) connecting the hidden layer to the output.

Step 2: Proportional Responsibility (w_{ho})

How do we distribute "blame" to the previous layer? We use the weights as a guide. The error at an output node is split among its incoming links in **proportion to their magnitudes**:

- **Larger Weights:** Carry more signal, so they receive a **larger share** of the error.
- **Smaller Weights:** Carry less signal and receive a **smaller share**.

This ensures that the paths that contributed most to the final output are the ones that receive the most significant adjustments.

Step 3: Inferred Error for the Hidden Layer (e_{hidden})

Hidden nodes do not have target values provided by training data. Instead, their error is **inferred** from the output layer.

We calculate e_{hidden} by **recombining the errors** from the output nodes that the hidden node influences, weighted by the strength of the connections:

$$e_{\text{hidden}} = \sum(\text{Weights} \times e_{\text{output}})$$

The Chain Reaction: This process repeats for the input-to-hidden weights (w_{ih}), allowing the error to flow from right to left through the entire architecture.

Inferred Error: Analyzing the Hidden Layer Nodes

Since hidden nodes lack direct target values, we calculate their error by "inheriting" a portion of the mistakes made at the output layer.

1. Hidden Node 1 ($e_{\text{hidden},1}$) This node influences both outputs via weights w_{11} and w_{12}. Its error is the sum of the proportional shares

it contributed to each output:

$$e_{\text{hidden},1} = \underbrace{e_{\text{output},1}\left(\frac{w_{11}}{w_{11}+w_{21}}\right)}_{\text{Responsibility for Output 1}} + \underbrace{e_{\text{output},2}\left(\frac{w_{12}}{w_{12}+w_{22}}\right)}_{\text{Responsibility for Output 2}}$$

2. Hidden Node 2 ($e_{\text{hidden},2}$) Similarly, the second hidden node inherits its error through weights w_{21} and w_{22}. Notice how the denominator remains the sum of weights connected to that specific output:

$$e_{\text{hidden},2} = \underbrace{e_{\text{output},1}\left(\frac{w_{21}}{w_{11}+w_{21}}\right)}_{\text{Responsibility for Output 1}} + \underbrace{e_{\text{output},2}\left(\frac{w_{22}}{w_{12}+w_{22}}\right)}_{\text{Responsibility for Output 2}}$$

Why this works:

- **Total Accountability:** For any output node (e.g., Output 1), the sum of the fractions distributed back to the hidden nodes always equals 1.0 (e.g., $\frac{w_{11}+w_{21}}{w_{11}+w_{21}}$).
- **Gradient Flow:** This proportional split creates the **gradient signal** used to update the weights between the input and hidden layers (w_{ih}).

By calculating these "inferred" errors, we allow the learning signal at the final layer to penetrate deep into the network, refining every internal connection.

Weight Update Rule: Hidden to Output Layer

The Learning Step: Updating Weights (w_{jk})

Once the error signals have been calculated, we perform the actual "learning" by adjusting each weight from hidden node j to output node k:

$$w_{j,k}^{(2)} \leftarrow w_{j,k}^{(2)} + \Delta w_{j,k}^{(2)}$$

The adjustment Δw is determined by the following formula:

$$\Delta w_{j,k}^{(2)} = \eta \cdot \delta_k^{(2)} \cdot h_j$$

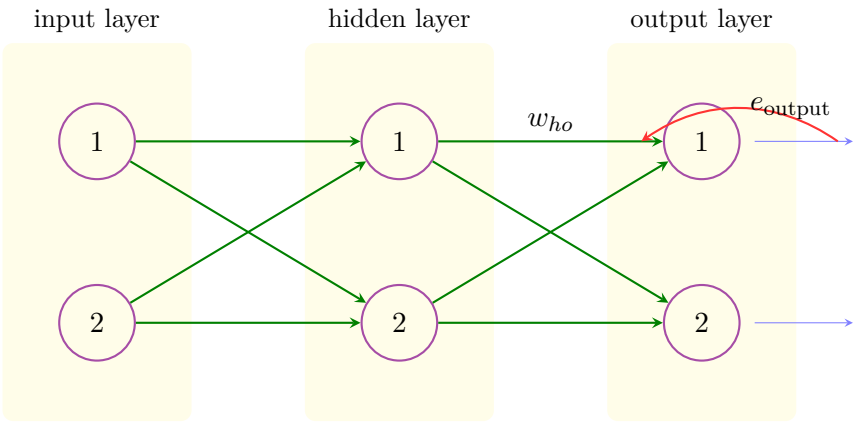


Figure 5.26: Backpropagating error from the output layer to the hidden layer.

The Three Factors of Change: This update strengthens or weakens a connection based on three specific variables:

- **The Learning Rate (η):** Controls the size of the step we take. A small η ensures stability, while a large η speeds up training but risks overshooting.
- **The Delta Term ($\delta_k^{(2)}$):** Combines the *output error* and the *slope of the sigmoid*. It determines the direction and necessity of the change.
- **Activation Strength (h_j):** The signal from the hidden layer. If a hidden neuron was "silent" ($h_j \approx 0$), it cannot be blamed for the error, so its outgoing weight won't change.

AI Insight: This is the realization of Gradient Descent. We are moving the weight in the opposite direction of the error gradient to reach a local minimum.

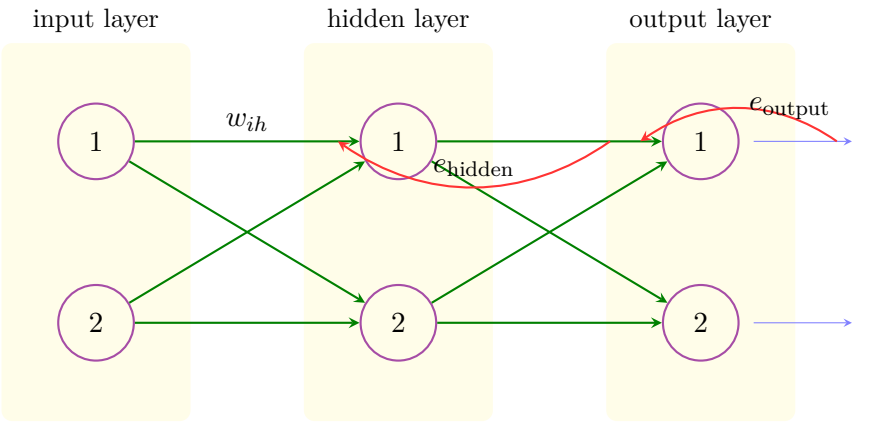


Figure 5.27: Errors are propagated further back from the hidden layer to the input layer.

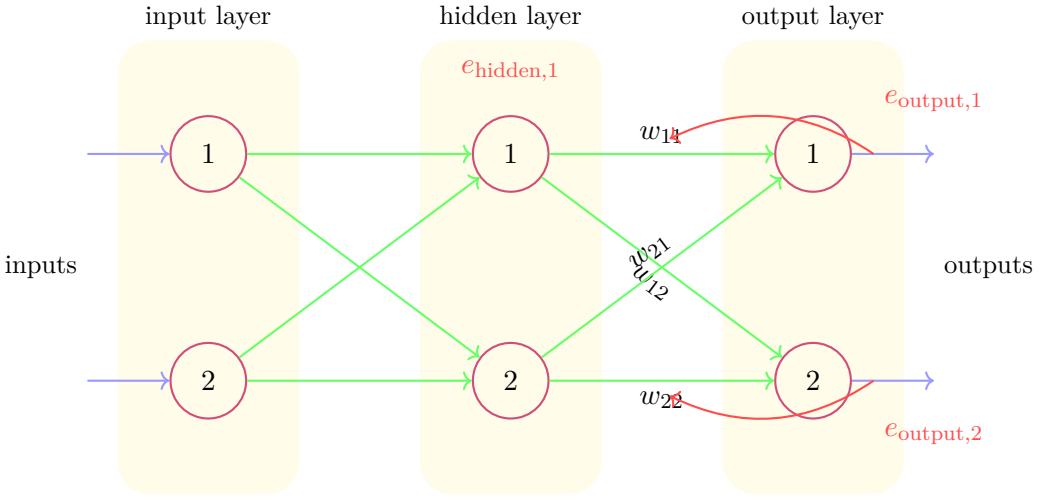


Figure 5.28: Backpropagation of errors from the output layer to the hidden layer. The hidden layer error is formed by recombining proportionally split output errors.

Propagating Error to the Hidden Layer

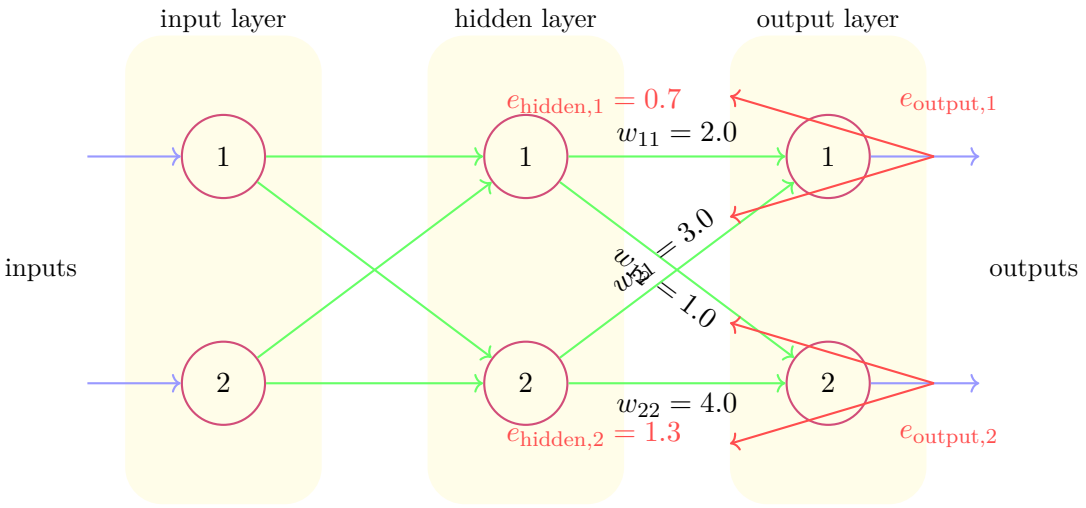


Figure 5.29: Forward signal flow (green) and backward propagation of output errors (red). The backpropagation arrows terminate between the hidden and output layers, indicating error contribution to the weighted connections.

Hidden Layer Gradient: The Delta Term ($\delta_j^{(1)}$)

Since hidden-layer neurons do not have direct access to target values, we compute their "responsibility" by backpropagating the errors from the layer ahead:

$$\delta_j^{(1)} = h_j(1 - h_j) \sum_{k=1}^{3} w_{j,k}^{(2)} \delta_k^{(2)}$$

The Hidden Neuron's Responsibility depends on:

- **Influence:** How strongly the hidden node influences the output neurons (determined by the weights $w_{j,k}^{(2)}$).
- **Output Error:** How "wrong" those specific output neurons were ($\delta_k^{(2)}$).
- **Local Gradient:** The derivative of the hidden node's own activation, $h_j(1 - h_j)$.

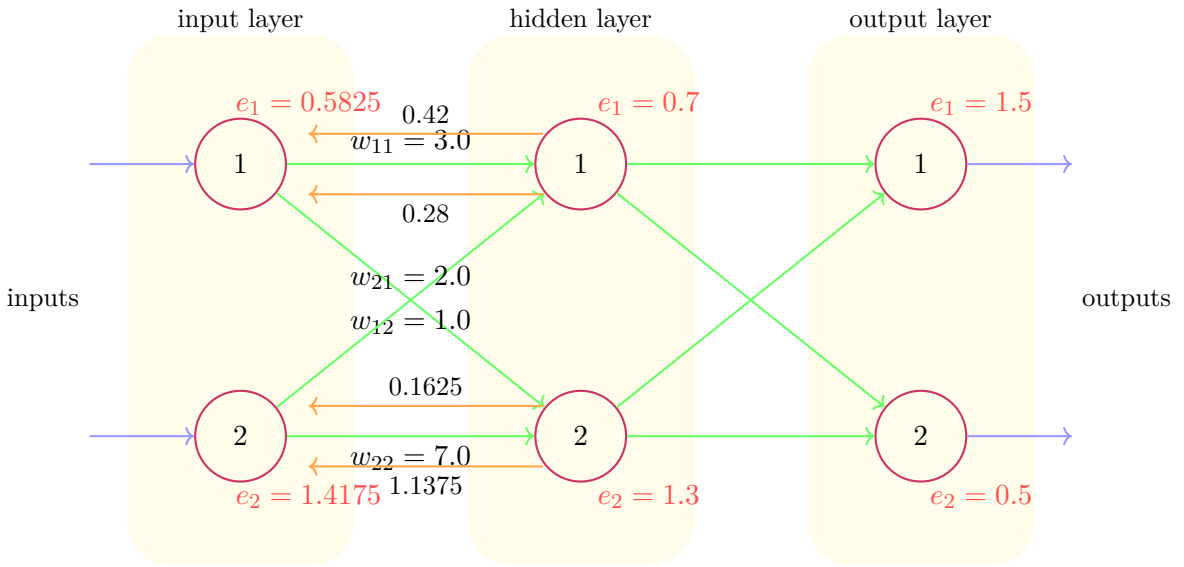


Figure 5.30: Backpropagation of error from output to hidden and further to input layer. Each input node error is formed by recombining the split errors from the hidden layer connections.

The Final Update: Input to Hidden Weights

Closing the Loop: Updating Weights ($w_{i,j}^{(1)}$)

Finally, we update the weights connecting the **Input Layer** to the **Hidden Layer** using the learning rate η and the raw input signal x_i:

$$w_{i,j}^{(1)} \leftarrow w_{i,j}^{(1)} + \eta\, \delta_j^{(1)}\, x_i$$

The Complete Cycle: Learning has now propagated fully backward from the output layer, through the hidden layer, and finally to the input weights. This completes one iteration of the Backpropagation algorithm.

Key Insight

Learning in a three-layer neural network occurs by:

1. computing output errors,
2. propagating those errors backward,

3. and updating each weight in proportion to its contribution.

Through repeated exposure to training data, the network gradually refines its weights, enabling accurate classification and prediction even for complex, non-linear problems.

At this point, we have developed a complete and efficient mathematical framework for training neural networks. Take a well-deserved break = the final theory section builds on these ideas and is especially rewarding.

Chapter 6

Deep Learning Architectures

Overview

Deep learning extends artificial neural networks by introducing architectures specifically designed to handle complex, high-dimensional data. This chapter explores the fundamental frameworks that power modern AI, moving from spatial recognition to temporal sequence modeling.

We will examine three cornerstone architectures:

- **Convolutional Neural Networks (CNNs)**: Optimized for grid-like data such as images.
- **Recurrent Neural Networks (RNNs)**: Designed for sequential logic and time-series.
- **Long Short-Term Memory Networks (LSTMs)**: Advanced units for long-term dependency retention.

Learning Objectives

By the end of this chapter, you will be able to:

- Understand why fully connected neural networks are inefficient for image data.
- Explain the intuition behind convolution and feature extraction.
- Perform convolution operations manually using Examples.
- Understand the role of filters, feature maps, stride, and padding.
- Interpret CNN operations visually and mathematically.

Introduction to the CNN Architectural Flow

The journey of data through this network begins at the **Input Layer**, where a raw color image is represented as a **32x32x3 tensor**, signifying the height, width, and the three primary RGB color channels. This input is immediately processed by a **Convolutional (CONV) Layer** consisting of **16 distinct learnable filters**, each with a **3x3x3 spatial extent**. As these filters convolve across the image with a **stride of 1** and **no padding (Valid Padding)**, they perform element-wise multiplications and summations that reduce the spatial dimensions to **30x30** while simultaneously increasing the depth to **16**, effectively creating a set of **ReLU-activated feature maps**. This specific transformation involves a total of **448 parameters**, calculated from the weights of the filters plus their respective biases. To streamline these features and enhance the model's robustness, the data enters a **Max Pooling Layer** which utilizes a **2x2 window** and a **stride of 2**. This operation aggressively downsamples the spatial resolution by half, resulting in a **15x15x16 pooled output** that retains only the most critical activation values. Following this feature extraction phase, the 3D volume undergoes **Flattening**, a process that rearranges the values into a single **3600-dimensional Feature Vector**. This high-dimensional representation is finally fed into a **Softmax (σ) activation function**, which normalizes the scores into a **probability distribution**, allowing the network to output a final prediction based on the learned characteristics of the original input image.

Figure 6.1: Numerical walkthrough of a CNN architecture: From a $32^2 \times 3$ RGB input to a 3600-dimensional flattened vector for classification.

By: Engr. Dr. Muhammad Siddique, Postdoc AI (YALE University, USA).
HOD Artificial Intelligence, NFC IET Multan, msiddique@nfciet.edu.pk

6.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a specialized class of artificial neural networks that are particularly well suited for processing data with a grid-like structure, such as images. Over the last decade, CNNs have become the foundation of modern computer vision systems and are widely used in applications such as image classification, object detection, face recognition, medical image analysis, handwriting recognition, and autonomous driving.

Traditional fully connected neural networks struggle when dealing with high-dimensional inputs like images. CNNs overcome these limitations by exploiting the spatial structure present in image data. By learning local patterns and reusing the same parameters across the entire image, CNNs achieve both computational efficiency and strong generalization performance.

This chapter develops a complete understanding of Convolutional Neural Networks starting from intuition and motivation, and gradually moving toward mathematical formulation and Examples.

6.1.1 The Architecture of a Neural Network Classifier

The neural network architecture depicted is a **Fully Connected Feedforward Network** designed to transform raw input data into specific categorical classifications (cat or dog). Unlike the raw pixel-processing of a CNN, this structure focuses on the weighted relationships between individual neurons across several stages.

There are three primary components to this classification network:

- **Input Layer:** This layer receives the initial data. In this context, it represents the features extracted from the bird images (such as plumage color, beak shape, or size) and passes them into the network.

- **Hidden Layers:** These intermediate layers perform the "learning." Each connection has an associated weight that adjusts as the network is trained. Multiple hidden layers allow the network to capture complex, non-linear patterns to distinguish between the bird varieties.

- **Output Layer:** The final layer maps the processed information into discrete categories. In this binary classification task, the nodes correspond to the probability of the input being a *cat* or a *dog*.

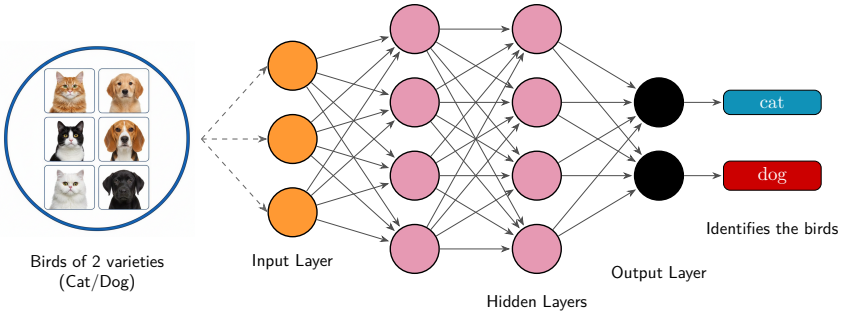


Figure 6.2: Scalable Deep Learning Architecture for Bird Classification.

6.1.2 Overview of the image classification using CNN

Binary Translation of the Cat Image

$$\text{cat} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

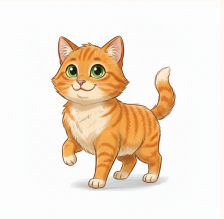


Figure 6.3: Original Image

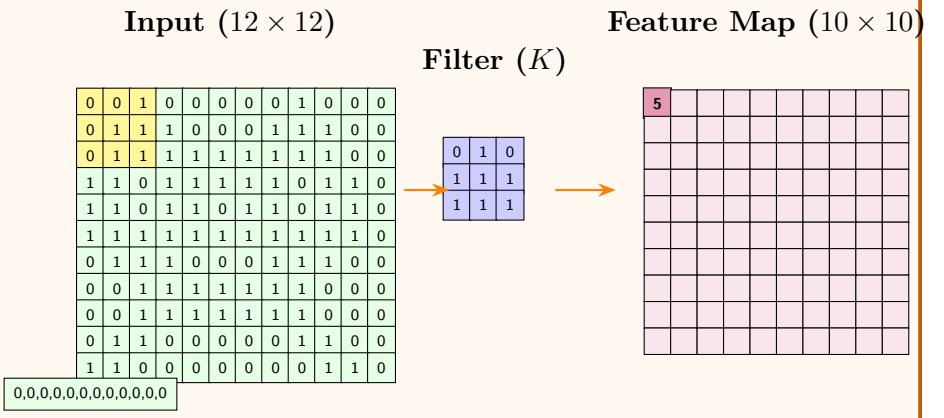
Cat Ear Detection Filter (K_{ear})

To identify the ears within the 12×12 binary grid, we apply a 3×3 feature kernel designed to match the "pointy" spatial arrangement of an ear tip:

$$K_{ear} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Target Feature: Ear

Mechanics of the Sliding Filter:
Below, the K_{ear} filter convolving over the 12×12 grid results in a 10×10 Feature Map.



Feature Map Extraction: The output value **5** is obtained by the Dot Product of the 3×3 window and the kernel:

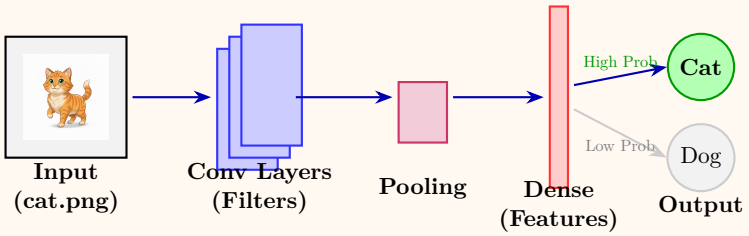
$$(0 \cdot 0 + 0 \cdot 1 + 1 \cdot 0) + (0 \cdot 1 + 1 \cdot 1 + 1 \cdot 1) + (0 \cdot 1 + 1 \cdot 1 + 1 \cdot 1) = 0 + 2 + 3 = \mathbf{5}$$

A high value relative to the maximum possible score (8) indicates a strong match for the "ear" feature.

Cat Ear Detection Filter (K_{ear})

How Classification Works: The CNN uses these activation values to distinguish between classes:

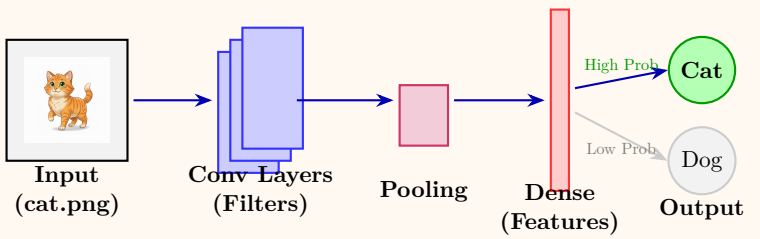
- **Spatial Patterns:** The network looks for high values in the "Ear" maps in the top corners.
- **Class Voting:** In the final Fully Connected Layer, a **Cat** is predicted if high activations are found for "Pointy Ears" and "Whiskers." A **Dog** is predicted if "Pointy Ear" filters show low activation but "Floppy Ear" or "Square Snout" filters show high activation.



Cat Ear Detection and CNN Classification Pipeline

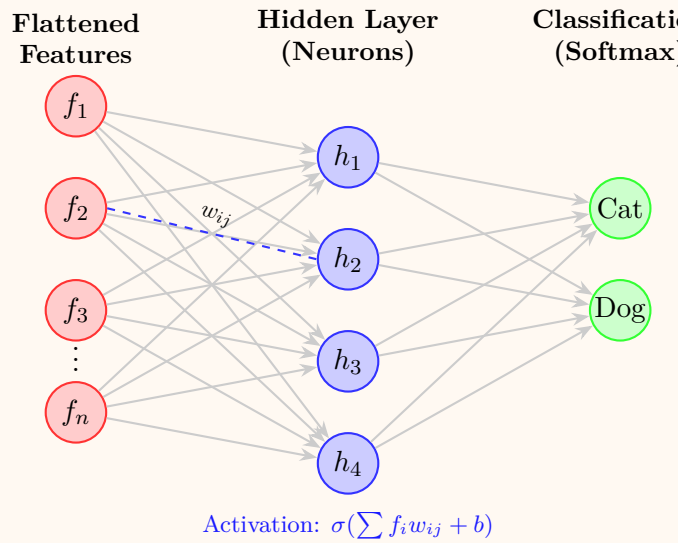
The Full Classification Journey (Missing Steps): The architectural diagram below illustrates the high-level flow. However, to transform the raw convolution output into a "Cat" vs "Dog" decision, several critical steps occur behind the scenes:

1. **ReLU Activation:** Immediately after the convolution shown above, the network applies a *Rectified Linear Unit* function. This suppresses negative values and allows the model to learn non-linear features.
2. **Pooling:** As seen in the **Pooling** block, the 10×10 maps are downsampled (typically to 5×5). This reduces computational load and makes the detection robust to small shifts in the image.
3. **Flattening:** Before reaching the **Dense** layer, the 2D feature maps are "unrolled" into a long 1D vector. This bridge allows spatial features to be interpreted as numerical inputs for the final decision.
4. **Softmax Output:** The final **Cat** and **Dog** nodes use a *Softmax* function. This converts raw "votes" into probabilities (e.g., 98% Cat), which is depicted by the "High Prob" arrow in the figure.



Internal Structure of the Dense (Fully Connected) Layer

The Dense Layer acts as the "brain" of the CNN. After the Convolutional layers extract spatial features (like the cat's ear), the Dense layer interprets these features globally to perform classification.



Internal Structure of the Dense (Fully Connected) Layer

Key Mathematical Components:

- **Flattening:** The 10×10 feature map is converted into a 1D vector $f = [f_1, f_2, \ldots, f_{100}]$.
- **Weights (w_{ij}):** These represent the "strength" of the connection. For example, if f_1 corresponds to an "ear" pixel, its weight w to the "Cat" neuron will be high.
- **Bias (b):** An additional parameter that allows the neuron to adjust its threshold for firing.
- **Softmax Activation:** The final layer converts raw scores into probabilities. For a cat image, the output might be $P(\text{Cat}) = 0.98$ and $P(\text{Dog}) = 0.02$.

6.2 Image Representation in Neural Networks

The process of digitizing an image involves transforming a visual signal into a numerical format that a computer can process. As shown in the diagram, this transformation typically follows three stages:

- **Real Image:** The original input, such as a handwritten digit '8', exists as a continuous visual form.
- **Array Representation:** To analyze the image, a grid or spatial array is overlaid. This segments the image into discrete units called pixels.
- **Binary Pixel Matrix:** Each cell in the grid is assigned a numerical value. In a simplified binary model, a '0' represents background space (white), while a '1' represents the presence of the digit (black). This matrix serves as the actual input volume for the neural network layers.

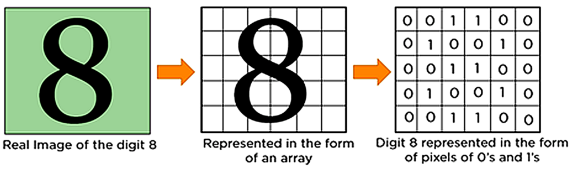


Figure 6.4: digital format of image

6.2.1 From Light to Digital Image Representation

Step 1: Light Interaction with the Real-World Scene

All visual information begins with light. Natural or artificial light sources illuminate objects in the environment. Objects reflect, absorb, or emit light depending on their physical properties.
The reflected light encodes:

- Intensity (brightness),
- Wavelength (color),
- Spatial structure of the scene.

This reflected light carries the raw visual information that will later be transformed into digital data.

Step 2: Optical Focusing by the Camera Lens

The camera lens collects incoming light rays and focuses them onto the image sensor. The lens forms a continuous two-dimensional projection of the three-dimensional world.
Key optical controls include:

- Focal length (field of view),
- Aperture (amount of light),
- Optical sharpness.

At this stage, the image is still continuous and purely optical.

Step 3: Exposure Control and Photon Integration

The shutter mechanism controls how long the sensor is exposed to incoming light. The total light energy captured by the sensor is determined by exposure.

$$\text{Exposure} = \text{Light Intensity} \times \text{Exposure Time}$$

This step controls image brightness and motion effects but does not yet produce digital data.

Step 4: Conversion of Light into Electrical Signals

The focused light falls onto the image sensor (CCD or CMOS), which consists of millions of photodiodes arranged in a grid.
Each photodiode:

- Absorbs incoming photons,
- Releases electrons via the photoelectric effect,
- Accumulates an electrical charge proportional to light intensity.

Thus, brightness information is converted into analog electrical signals at each pixel.

Step 5: Color Sampling Using a Color Filter Array

Image sensors inherently measure light intensity, not color. To capture color information, a Color Filter Array (CFA), commonly the Bayer pattern, is placed over the sensor.

$$\begin{bmatrix} G & R & G \\ B & G & B \\ G & R & G \end{bmatrix}$$

Each pixel records only one color component (Red, Green, or Blue). Missing color values are reconstructed computationally, forming a full-color image.

Step 6: Analog-to-Digital Conversion (Digitization)

The analog electrical signals produced by the sensor are converted into discrete numerical values using an Analog-to-Digital Converter (ADC).

- Continuous voltages $\rightarrow$ discrete levels,
- Typical precision: 8-bit or 12-bit per channel.

$$\text{Pixel Value} \in \{0, 1, 2, \ldots, 255\}$$

This step marks the transition from the physical world to the digital domain.

Step 7: Formation of the Digital Image Tensor

After digitization, the image is represented as a numerical array:

$$\mathbf{I} \in \mathbb{R}^{H \times W \times C}$$

where:

- H is the image height,
- W is the image width,
- C is the number of channels (1 for grayscale, 3 for RGB).

Each element of this tensor corresponds to a pixel intensity value.

Step 8: Storage and CNN-Ready Representation

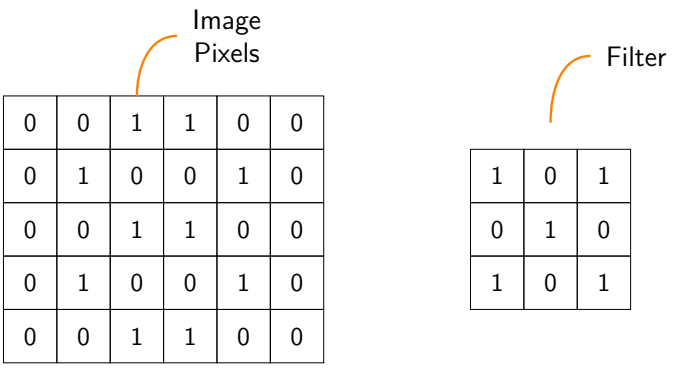
The digital image is stored in computer memory or disk and later loaded for processing. Before being input to a CNN, the image is typically normalized and resized.

$$\mathbf{X} = \frac{\mathbf{I}}{255}$$

A CNN processes this tensor purely as numerical data, learning edges, textures, and higher-level features through convolution operations.

6.2.2 The Filter for the Convolution

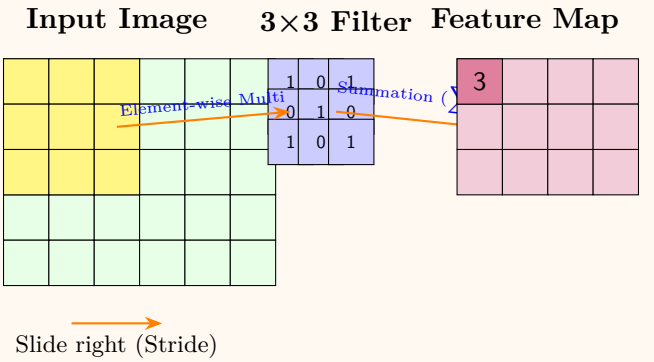
This is where the magic happens. The CNN uses this filter to perform the heavy lifting of feature extraction.



Mechanism of the Sliding Filter

Mechanics of the Sliding Filter

In a Convolutional Neural Network, the **filter** acts as a feature detector. The process of sliding this filter over the input image is what allows the network to maintain *spatial hierarchy*=understanding where a specific feature (like a wing or a beak) is located relative to others.



Mechanics of the Sliding Filter

- **The Receptive Field:** The yellow area represents the *local receptive field*. The filter only "looks" at this small patch at any given time, which mimics how human vision focuses on specific details before perceiving the whole object.
- **The Dot Product:** As the filter slides, it calculates the **sum of products**. If the pixel values in the yellow box align perfectly with the filter's weights, the output in the purple **Feature Map** will be a high number, indicating a "match" for that feature.
- **Stride and Translation Invariance:** The "Slide" (or **Stride**) determines how many pixels the filter moves. Because the same filter is used across the entire image, the CNN can recognize a "cat" whether it is in the top-left corner or the center of the image.
- **Feature Mapping:** The final purple grid is a condensed map of where specific features were detected. This map is then passed to deeper layers to identify higher-level structures, such as the difference between a sharp beak and a rounded head.

Mathematical Representation of Matrix Multiplication

Using the 5×6 binary image (I) representing a digit and the 3×3 filter (K) defined in the architectural discussion, the feature map is generated via element-wise multiplication and summation.

Input Image (I) **Filter (K)**

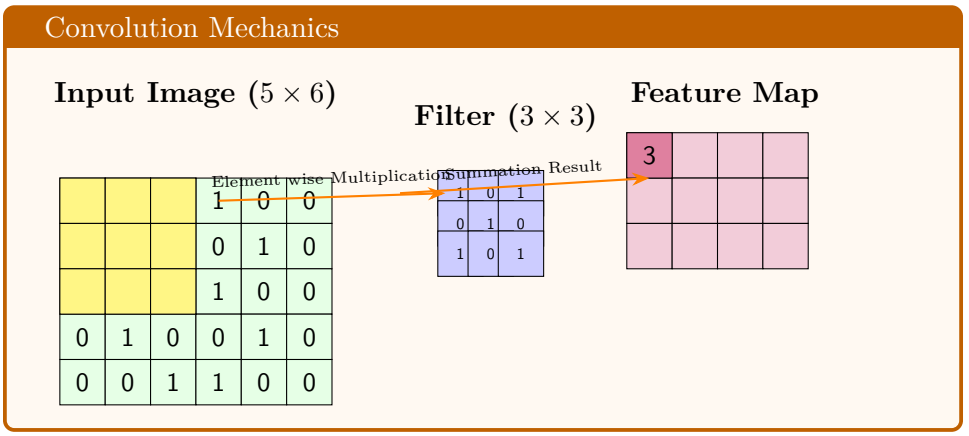
$$I = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \quad \text{and} \quad K = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

For the first position (top-left), we extract the 3×3 sub-matrix $Sub\_I_{(1,1)}$ and compute the dot product:

$$O_{1,1} = (0\times1)+(0\times0)+(1\times1)+(0\times0)+(1\times1)+(0\times0)+(0\times1)+(0\times0)+(1\times1) = 3$$

Visualizing the Extraction Process

The following diagram illustrates the relationship between the input pixels, the sliding filter, and the resulting feature map.



By: Engr. Dr. Muhammad Siddique, Postdoc AI (YALE University, USA).
HOD Artificial Intelligence, NFC IET Multan, msiddique@nfciet.edu.pk

Why Use a Specific Filter?

In a Convolutional Neural Network, a **filter** (or kernel) is used to perform feature extraction by identifying specific patterns within input pixels, such as the digit "8" or bird features.

Filter Example (3x3)

1	0	1
0	1	0
1	0	1

- **Pattern Recognition:** Filters are designed to detect unique features such as vertical edges, horizontal lines, or curves. When the filter slides over the image (like the 6×6 matrix of the digit "8"), it looks for matches between its own values and the underlying pixels.
- **Dimensionality Reduction:** By convolving across the image, the filter summarizes a local neighborhood into a single numerical value, effectively condensing spatial information.
- **Learnable Parameters:** These filters are not static; during training, the network learns the optimal values for these matrices to distinguish between different classes, such as identifying the ears of a *cat* versus a *Hen*.

Fully Connected Networks (ANNs) Are Not Suitable for Images

Fully connected neural networks are not well suited for image data due to their high computational cost and poor handling of spatial information.

Consider a grayscale image of size 32×32. Such an image contains

$$32 \times 32 = 1024$$

pixels. If this image is flattened into a vector and fed into a fully connected network with a single hidden layer of 100 neurons, the number of required weights becomes

$$1024 \times 100 = 102{,}400.$$

This large number of parameters arises from only one layer.

As image resolution increases, the parameter count grows rapidly. For example, a color image of size $224 \times 224 \times 3$ would require millions of parameters in a fully connected network. This leads to excessive memory usage, high computational cost, and a significant risk of overfitting.

In addition, fully connected networks ignore the spatial structure of images. Neighboring pixels that form meaningful patterns such as edges or textures are treated as independent inputs. As a result, the network cannot naturally learn that local pixel groups represent visual features or objects, regardless of their position in the image.

Convolutional Neural Networks (CNNs) overcome these limitations by preserving spatial relationships, using local receptive fields, sharing weights across the image, and learning hierarchical visual features.

Example:

Convolution is a mathematical operation that combines two pieces of information: an input image and a small matrix called a **filter** or **kernel**. The filter slides across the image and computes a weighted sum of pixel values at each location. Each position produces a single number, and all such numbers together form a new image called a **feature map**.

Intuitively, the filter acts as a pattern detector. Different filters detect different features such as edges, corners, or textures. When the pattern in the image matches the pattern in the filter, the convolution results in a high value in the feature map.

Consider the following 4×4 input image $\mathbf{I}$ and the 2×2 filter $\mathbf{K}$:

$$\mathbf{I} = \begin{bmatrix} 1 & 2 & 0 & 1 \\ 3 & 1 & 2 & 2 \\ 0 & 1 & 3 & 1 \\ 2 & 2 & 1 & 0 \end{bmatrix} \quad \mathbf{K} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

We place the filter on the top-left corner of the image and compute the element-wise product and sum:

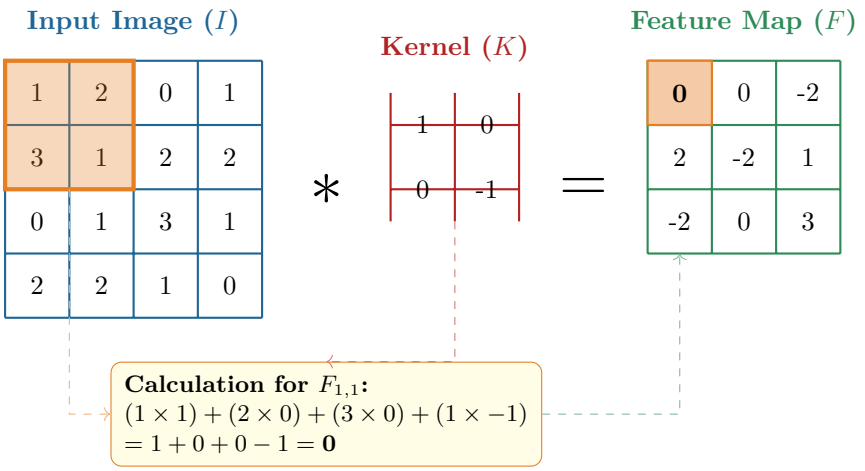
$$(1 \times 1) + (2 \times 0) + (3 \times 0) + (1 \times -1) = 1 + 0 + 0 - 1 = \mathbf{0}$$

Next, the filter is shifted one position to the right (Stride = 1) and the computation is repeated:

$$(2 \times 1) + (0 \times 0) + (1 \times 0) + (2 \times -1) = 2 + 0 + 0 - 2 = \mathbf{0}$$

This process continues across the image. After completing the first row, we shift down one row and start from the left again. The resulting feature map $\mathbf{F}$ is:

$$\mathbf{F} = \begin{bmatrix} 0 & -2 & -2 \\ 2 & -2 & 1 \\ -2 & 0 & 3 \end{bmatrix}$$



This feature map highlights locations where the pattern encoded by the filter appears strongly in the image. Through this simple mathematical mechanism, CNNs are able to transform raw pixel data into meaningful visual features.

The efficacy of CNNs stems from several unique mathematical properties that distinguish them from standard neural architectures:

- **Feature Specialization**: Each filter in a layer is initialized differently, allowing it to "specialize" in a specific visual cue, such as a vertical edge or a specific color gradient.
- **Weight Sharing**: Unlike fully connected networks, the same filter weights are applied to every patch of the image. This drastically reduces the parameter count and provides *translation invariance*.
- **Spatial Preservation**: Convolution maintains the relative positions of pixels, ensuring that the spatial orientation of features is not lost.
- **Filter Depth**: Modern CNNs use hundreds of filters per layer, producing a "stack" of feature maps that represent the image in a high-dimensional feature space.

6.3 Properties of Convolution

1. Stride

Stride defines the "step size" of the filter as it traverses the image. A **Stride of 1** moves the filter one pixel at a time, resulting in a dense, overlapping scan. Conversely, a larger stride (e.g., **Stride of 2**) skips pixels, effectively performing *downsampling* by reducing the spatial resolution of the resulting feature map.

2. Padding

Padding is the technique of appending extra pixels=usually with a value of zero=around the perimeter of the input image. This is referred to as **Zero-Padding**. Padding is critical for two functional reasons:

- **Preventing Dimensional Shrinkage**: Without padding, the output of a convolution is always smaller than the input. In deep networks with dozens of layers, the feature map would vanish entirely without padding.

- **Border Information Retention**: Pixels at the edges of an image are only touched by the filter a few times, whereas central pixels are covered many times. Padding allows the filter to "reach" beyond the edges, ensuring that information at the boundaries is treated with equal importance.

Padding is not merely a buffer in convolutional architectures; it is a structural necessity for deep learning models. Without padding, each convolution operation progressively removes boundary pixels, causing the spatial dimensions of feature maps to shrink and eventually disappear in deep networks. Padding preserves critical boundary information by ensuring that edge pixels contribute equally to convolution operations, and it enables the construction of very deep architectures by maintaining consistent feature map sizes across layers. As a result, padding allows convolutional neural networks to learn complex, high-level representations without losing spatial context or global structure.

Zero-Padding (p=1)

3. Spatial Dimensions

Predicting the spatial dimensions of a layer is a fundamental skill in CNN architecture design. The output size O is governed by the input size N, the filter size F, the padding P, and the stride S:

$$O = \left\lfloor \frac{N-F+2P}{S} \right\rfloor + 1$$

Example: The "Same" Padding Effect

Let us calculate the output size for a common configuration used to maintain image resolution:

- **Input** (N): 32
- **Filter** (F): 5
- **Padding** (P): 0
- **Stride** (S): 1

Substituting into the formula:

$$\text{Output size} = \frac{32 - 5 + 2(0)}{1} + 1 = \frac{27}{1} + 1 = 28$$

As shown, a padding of 0 with a 5×5 filter results in an output size that matches the input reduction seen in standard convolution. This is known as **"Valid Padding"** and is used extensively in deep architectures like ResNet to prevent the feature maps from shrinking too early in the network.

4. Pooling

While convolutional layers focus on feature extraction, **Pooling Layers** (also known as downsampling or subsampling) focus on **spatial efficiency**

and **robustness**. As we move deeper into a CNN, we want to decrease the resolution of our feature maps while increasing the number of filters.

Pooling acts as a "summarization" mechanism. By focusing on the most prominent values within a window, the network discards the exact pixel-perfect location of a feature in favor of its relative position. This makes the model resilient to minor distortions or shifts in the input image.

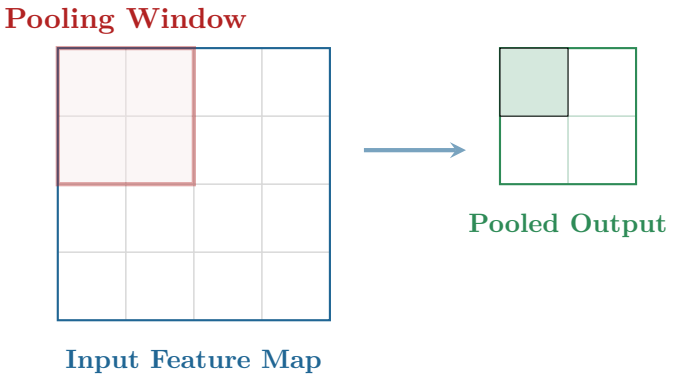


Figure 6.5: The Max Pooling operation: Reducing spatial resolution while preserving the most salient features.

The primary objective of pooling is to reduce the number of parameters and computational cost in the network. There are two dominant types of pooling:

Max Pooling: Extracting the Dominant Signal

Max pooling is the industry standard. It slides a window (typically 2×2) over the feature map and selects only the **maximum value** within that window. This approach is based on the logic that the highest value represents the strongest presence of a feature in that local region.

Conceptual Advantages of Max Pooling:

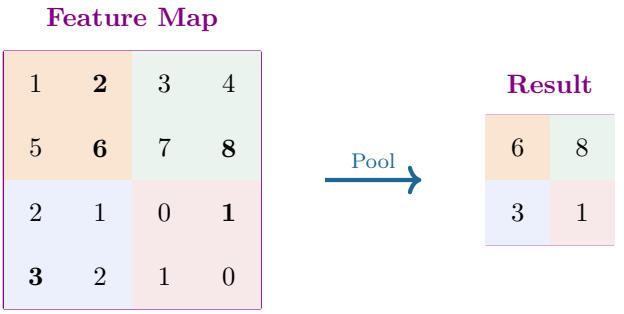
- **Translation Invariance**: Small shifts in the input image do not change the output of max pooling, making the network more robust to object movement.
- **Dimensionality Reduction**: A 2×2 pool with a stride of 2 reduces the total number of pixels by **75%**.
- **Overfitting Control**: By reducing the data resolution, the model is forced to learn more general features rather than memorizing exact pixel locations.

Average Pooling: Capturing Global Context

Average pooling calculates the mean value of the pixels within the window. While it was common in early architectures, it is now largely used in the final layers of modern networks (Global Average Pooling) to condense all spatial information into a single value before classification.

Example of Max Pooling

To understand the mechanics, consider a 4×4 feature map where we apply 2×2 max pooling with a **stride of 2**.



In this example, the spatial resolution is halved. Note how the most "salient" or high-value features (6, 8, 3, and 1) are preserved in their respective quadrants, while less significant details are discarded. This allows the network to maintain its "memory" of the feature's existence without needing to track its exact pixel-perfect coordinate.

6.3.1 Flattening and Classification

The final stage of a CNN architecture is the transition from spatial feature maps to a classification decision. Once the feature maps have been reduced

to a sufficiently small spatial size (e.g., 7×7), they are **Flattened**.

Flattening unrolls the 3D tensor into a 1D vector. This vector is then passed into one or more **Fully Connected (Dense) Layers**, which act as the final reasoning engine to map the learned features to specific class labels (using Softmax for probability).

Example:

Consider the final convolutional output volume of a modern network:

- **Tensor Dimensions**: $7 \times 7 \times 64$
- **Mathematical Logic**: We have 64 different feature maps, each of size 7×7.
- **Flattened Vector Length**: $7 \times 7 \times 64 = \mathbf{3,136}$ neurons.

This vector is now ready to be processed by a standard Artificial Neural Network (ANN) to produce the final output.

Note: The "Convolutional Base" acts as a feature extractor that understands the "What" and "Where" of the image, while the "Dense Head" acts as a classifier that interprets those findings to make a final prediction.

Case Study: A Simple CNN Architecture

To illustrate these concepts, consider a CNN designed to classify $32 \times 32 \times 3$ RGB images into 10 distinct categories. The process begins at the **Input Layer** with raw pixel intensities, which are first processed by **Conv Block 1** using 16 filters (3×3) and ReLU activation to capture low-level edges, then downsampled via **Max Pooling 1** to a 16×16 spatial size. A second stage, **Conv Block 2**, utilizes 32 filters to identify complex textures before **Max Pooling 2** further reduces the dimensions to 8×8. These spatial features are then subjected to **Flattening**, transforming the $8 \times 8 \times 32$ tensor into a 1D vector of length 2,048, which feeds into a **Dense Layer** of 128 neurons for non-linear reasoning. Finally, the **Output Layer** employs 10 neurons with **Softmax** activation to generate the final classification probabilities.

By: Engr. Dr. Muhammad Siddique, Postdoc AI (YALE University, USA).
HOD Artificial Intelligence, NFC IET Multan, msiddique@nfciet.edu.pk

6.4 Case Study: A Simple CNN Architecture

To illustrate, let us define a CNN designed to classify 32×32 RGB images (such as those in the CIFAR-10 dataset) into 10 distinct categories.

Architecture Specifications

1. **Input Layer**: A $32 \times 32 \times 3$ tensor representing the raw pixel intensities.
2. **Conv Block 1**: 16 filters (3×3) + ReLU. This captures low-level edges.
3. **Max Pooling 1**: 2×2 window. Reduces spatial size to 16×16.
4. **Conv Block 2**: 32 filters (3×3) + ReLU. Captures complex textures.
5. **Max Pooling 2**: 2×2 window. Reduces spatial size to 8×8.
6. **Flattening**: Converts the $8 \times 8 \times 32$ tensor into a 1D vector of length $2,048$.
7. **Dense Layer**: 128 neurons to perform non-linear reasoning on the features.
8. **Output Layer**: 10 neurons with **Softmax** activation for final classification.

6.4.1 Visualizing the Structural Flow

The following diagram represents the progressive "compression" of spatial information and the "expansion" of feature depth.

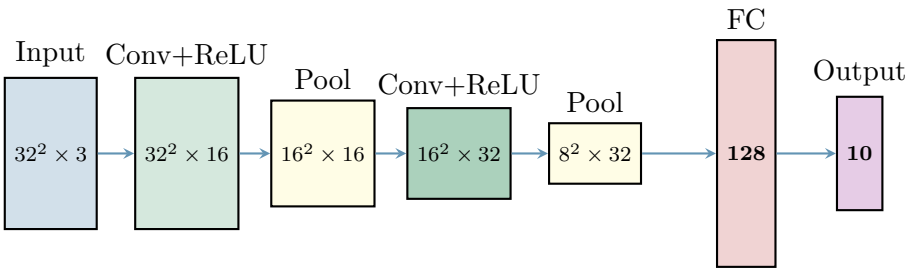


Figure 6.6: Evolution of data through a CNN: Spatial resolution decreases while the number of feature channels (depth) increases.

6.5 Some more topics related to CNN

6.5.1 The Decision Layer: Softmax Activation

For multi-class classification, we require the network to output a probability distribution. This is achieved using the **Softmax Function**. Given a raw output vector $\mathbf{z}$ (logits), Softmax normalizes the values:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

This ensures that every output $\hat{y}_i$ is between 0 and 1, and the sum of all outputs $\sum \hat{y}_i = 1$. Mathematically, this allows us to interpret the network's output as the **confidence level** for each class.

6.5.2 Loss Functions: Categorical Cross-Entropy

Training a CNN is an optimization problem: we want to minimize the error between the prediction $\hat{\mathbf{y}}$ and the ground truth $\mathbf{y}$. For multi-class tasks, we utilize **Categorical Cross-Entropy Loss**.

$$\mathcal{L} = -\sum_{i=1}^{K} y_i \log(\hat{y}_i)$$

Example

Suppose we are classifying an image into 3 categories (Cat, Dog, Bird). The true label is "Dog" ($y = [0, 1, 0]$), and our CNN predicts:

$$\hat{\mathbf{y}} = [0.1(\text{Cat}), \mathbf{0.7}(\text{Dog}), 0.2(\text{Bird})]$$

The loss is calculated as:

$$\mathcal{L} = -(0 \cdot \log(0.1) + \mathbf{1} \cdot \log(\mathbf{0.7}) + 0 \cdot \log(0.2)) = -\log(0.7) \approx 0.357$$

A perfect prediction ($\hat{y}_{\text{Dog}} = 1.0$) would result in a loss of 0. This loss value is backpropagated through the network to update the filter weights using **Stochastic Gradient Descent (SGD)**.

6.5.3 Training a CNN: The Forward and Backward Pass

The learning process of a Convolutional Neural Network is an iterative optimization cycle. By repeatedly exposing the model to data and correcting its errors, the filters evolve from random noise into specialized feature detectors. This cycle consists of two fundamental phases:

1. Forward Propagation: Feature Synthesis

During the **Forward Pass**, the input image is transformed into a high-level abstraction through a series of linear and non-linear operations.

- **Transformation**: The image propagates through the sequences of $Convolution \rightarrow ReLU \rightarrow Pooling$.
- **Inference**: The flattened features are processed by the Dense layers to produce raw scores (logits).
- **Probabilistic Mapping**: Softmax converts scores into probabilities, and the **Loss Function** $\mathcal{L}$ quantifies the discrepancy between the prediction and the ground truth.

2. Backward Propagation: Error Attribution

The **Backward Pass** (Backpropagation) is the mechanism by which the network "blames" specific weights for the resulting error.

- **Chain Rule**: Using the multivariable chain rule, the gradient of the loss $\frac{\partial \mathcal{L}}{\partial \hat{y}}$ is propagated backward from the output layer to the very first convolutional filter.
- **Spatial Gradient Flow**: As gradients pass through pooling layers, they are "upsampled" (in Max Pooling, the gradient only flows back to the neuron that had the maximum value during the forward pass).
- **Weight Updates**: The optimizer (e.g., SGD or Adam) adjusts every learnable parameter in the direction that minimizes the loss.

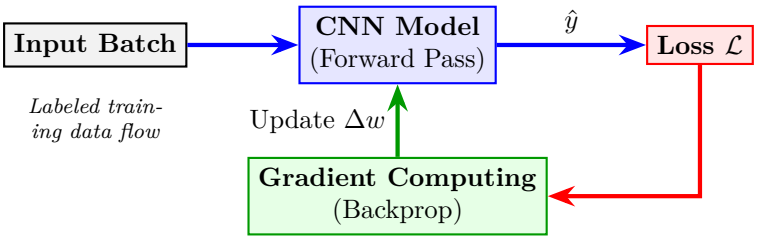
6.5.4 Backpropagation in Convolutional Layers

Backpropagation in CNNs follows the standard gradient descent logic, but it is uniquely characterized by **Weight Sharing**. Because the same filter is used across the entire image, the gradient for a single filter weight is the sum of gradients from every spatial location where that weight was applied.

For a specific filter weight w, the update rule is defined as:

$$w_{new} = w_{old} - \eta \frac{\partial \mathcal{L}}{\partial w}$$

where η represents the **learning rate**, a hyperparameter that controls the step size of the update.



Example:

Imagine a single 1×1 filter weight w applied to a 2×1 image $X = [x_1, x_2]$.

1. **Forward Pass:** The output $S = [s_1, s_2]$ is calculated as $s_1 = w \cdot x_1$ and $s_2 = w \cdot x_2$.
2. **Loss Calculation:** Suppose the network computes a loss $\mathcal{L}$. Through the chain rule, we find the "error" at the output pixels: $\frac{\partial \mathcal{L}}{\partial s_1} = 0.5$ and $\frac{\partial \mathcal{L}}{\partial s_2} = 0.3$.
3. **Weight Gradient (Accumulation):** Because w contributed to both s_1 and s_2, we sum the local gradients:

$$\frac{\partial \mathcal{L}}{\partial w} = \left(\frac{\partial \mathcal{L}}{\partial s_1} \cdot x_1\right) + \left(\frac{\partial \mathcal{L}}{\partial s_2} \cdot x_2\right)$$

   If $x_1 = 2$ and $x_2 = 4$, then $\frac{\partial \mathcal{L}}{\partial w} = (0.5 \cdot 2) + (0.3 \cdot 4) = 1.0 + 1.2 = \mathbf{2.2}$.
4. **Update:** With $\eta = 0.01$ and $w_{old} = 0.5$:

$$w_{new} = 0.5 - (0.01 \cdot 2.2) = \mathbf{0.478}$$

BS Student Insight: Unlike a standard ANN where each weight has its own dedicated gradient, a CNN filter weight w accumulates gradients from the entire image:

$$\frac{\partial \mathcal{L}}{\partial w} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial S_{i,j}} \cdot \frac{\partial S_{i,j}}{\partial w}$$

This **gradient accumulation** is why CNNs are so efficient=they learn general patterns (like edges) by looking at every part of the image simultaneously.

By: Engr. Dr. Muhammad Siddique, Postdoc AI (YALE University, USA).
HOD Artificial Intelligence, NFC IET Multan, msiddique@nfciet.edu.pk

6.5.5 Backpropagation Through Pooling Layers

A unique characteristic of pooling layers is that they contain **no learnable parameters** (weights or biases). Consequently, they do not "learn" in the traditional sense; however, they must still participate in the backward pass to propagate gradients to earlier convolutional layers.

- **Max Pooling**: During the forward pass, only the maximum value is passed forward. During backpropagation, the gradient flows **only** through the "winner" (the pixel that was the maximum), while the gradients for all other pixels in the window are set to zero.

- **Average Pooling**: The gradient is distributed **equally** among all pixels in the pooling window, as each pixel contributed equally to the average during the forward pass.

6.5.6 Modern Optimization Algorithms

While the foundation of training is **Gradient Descent**, modern CNNs require more sophisticated optimizers to navigate the high-dimensional, non-convex loss landscapes of deep learning.

1. Stochastic Gradient Descent (SGD) with Momentum

Standard SGD computes updates on small **mini-batches**. By adding a "momentum" term, the optimizer accumulates velocity in directions of persistent gradients, helping the model escape local minima and speed up convergence.

2. Adam (Adaptive Moment Estimation)

Adam is currently the most popular optimizer for CNNs. it computes **adaptive learning rates** for each individual parameter by tracking both the first moment (mean) and the second moment (uncentered variance) of the gradients. This makes it highly robust to noisy data and sparse gradients.

6.5.7 Combatting Overfitting in Deep CNNs

Because CNNs have millions of parameters, they are highly susceptible to **overfitting**=a state where the model memorizes the training noise rather than learning general patterns.

Dropout: Forced Redundancy

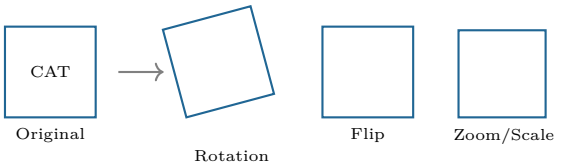
Dropout is a powerful regularization technique where, during each training step, a random subset of neurons is "dropped" (set to zero).

$$h_{dropped} = \mathbb{I}(\text{rand} > p) \cdot h$$

This prevents neurons from co-adapting too closely, forcing the network to learn multiple independent pathways to recognize the same feature.

Data Augmentation: Virtual Expansion

Since deep models are "data-hungry," we can artificially expand our dataset by applying label-preserving transformations.



6.5.8 Parameter Counting:

As an AI engineer, you must be able to audit the memory footprint of your model. The number of parameters in a convolutional layer depends **only** on the filter dimensions and the number of filters, not the input image size.

Calculation Rule

For a layer with N filters of size $F \times F$ and input depth D:

$$\text{Params} = N \times (F \times F \times D + 1)$$

(The +1 accounts for the bias term of each filter.)

Example: A layer with 32 filters of size 3×3 receiving an input with 16 channels:

$$32 \times (3 \times 3 \times 16 + 1) = 32 \times 145 = \mathbf{4,640} \text{ parameters.}$$

6.6 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) represent a paradigm shift in deep learning. Traditional Feedforward Neural Networks (ANNs) and Convolutional Neural Networks (CNNs) are essentially **amnesic**: they treat each input as

an isolated event, independent of what came before it. This "Independence Assumption" is highly effective for static data, such as identifying a cat in a single photograph. However, much of the data generated by the real world is inherently **sequential**.

In sequential data, the *order* of the observations is as critical as the observations themselves. RNNs address this by accounting for **temporal dependencies**, allowing information from the past to influence the processing of the current input.

Common domains where sequential logic is paramount include:

- **Natural Language Processing (NLP):** The meaning of a word is often dictated by the context of the preceding words.
- **Time-Series Analysis:** Financial markets or weather patterns exhibit trends where the current state is a direct consequence of historical fluctuations.
- **Speech Recognition:** Audio signals are continuous waves where the sound at millisecond t is physically linked to $t - 1$.
- **Bioinformatics:** DNA and protein sequences are "strings" of biological information where the arrangement determines functional properties.

By introducing an internal **hidden state** (memory), RNNs allow information to persist, effectively giving the network a "train of thought."

The Failure of Feedforward Architectures

Standard ANNs assume that the mapping from input to output $y = f(x)$ is purely a function of the current x. To see why this fails, consider the significance of syntax in language:

- **Sequence A:** *"The cat chased the mouse."*
- **Sequence B:** *"The mouse chased the cat."*

A feedforward network that processes these as a "bag of words" (unordered set) would see identical inputs for both. Because it lacks a sense of **temporal order**, it cannot distinguish between the predator and the prey. RNNs solve this by creating a **feedback loop**, where the network's previous internal state is fed back into itself alongside the new input.

6.7 The Recurrent Connection: A Simple Intuition

The core innovation of an RNN is the **hidden state (h_t)**. Think of the hidden state as a "summary" of everything the network has seen so far. At every tick of the clock (t):

1. The network looks at the new input (x_t).
2. It retrieves its previous memory (h_{t-1}).
3. It combines them to form a new, updated memory (h_t).

Insight: Memory as a Latent Variable

The hidden state is a *latent* (hidden) representation. It does not just store raw data; it stores the *features* and *context* required to make a prediction at the current or future time steps.

6.7.1 Visualizing RNN Architectures

1. Compact (Folded) Representation

The compact view is the "structural" blueprint. It shows the network as a single cell with a recursive connection.

2. Unrolling Through Time

To understand how backpropagation works, we "unroll" the loop. This reveals that an RNN is essentially a very deep neural network where each layer represents a different time step, but **all layers share the same weights**.

6.7.2 Mathematical Formulation

The behavior of the RNN at any given time t is governed by two fundamental equations.

The State Update Equation: The new hidden state is a non-linear combination of the current input and the previous state:

$$h_t = \phi\left(W_{xh}x_t + W_{hh}h_{t-1} + b_h\right)$$

The Output Equation: The output is generated directly from the current hidden state:

$$y_t = \psi\left(W_{hy}h_t + b_y\right)$$

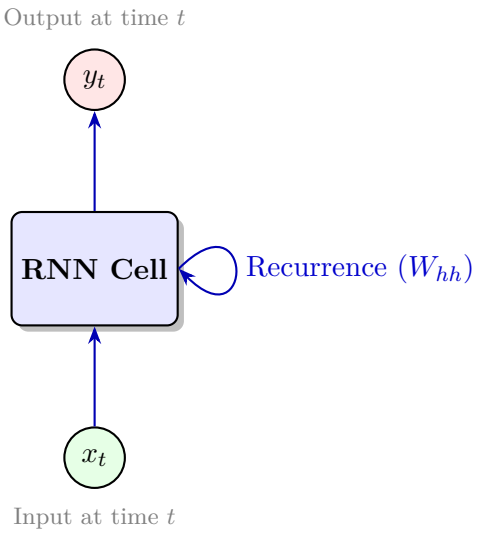


Figure 6.7: The folded representation highlights the recursive nature of the hidden state.

Component Breakdown

- **Weight Matrices (W_{xh}, W_{hh}, W_{hy}):** These are the trainable parameters. Note that W_{hh} is the "memory weight" that decides how much of the past to keep.
- **Activation Functions:** $\phi(\cdot)$ is almost always tanh or ReLU (to handle state transitions), while $\psi(\cdot)$ depends on the task (e.g., Softmax for classification).
- **Initial State (h_0):** Since there is no h_{-1}, h_0 is usually initialized as a vector of zeros or as a learned parameter.

Educational Note: Weight Sharing

Unlike a deep feedforward network where layer 1 and layer 2 have different weights, **RNNs use the same W matrices at every time step**. This allows the network to find the same pattern (like a verb following a noun) regardless of whether it happens at the start of a sentence or the end.

6.8 Example of an RNN:

Theoretical equations often hide the elegance of how RNNs actually "remember." To build a concrete intuition, we will trace the hidden state dynamics of

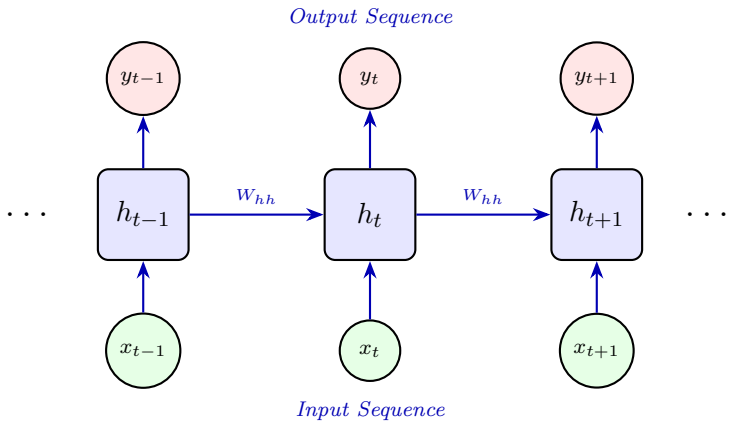


Figure 6.8: Unrolled representation of an RNN: the same network is applied at each time step with shared parameters.

a simple RNN as it processes a three-step numerical sequence. This demonstrates the transformation of raw inputs into a **temporal context**.

We define a "Vanilla" RNN with scalar components to keep the arithmetic transparent:

- **Input (x_t)**: A single numerical value at each time step.
- **State (h_t)**: A single value representing the internal memory.
- **Weights**: $W_{xh} = 0.6$ (Input-to-Hidden) and $W_{hh} = 0.8$ (Hidden-to-Hidden).
- **Activation**: $\phi = \tanh$, which squashes values between -1 and 1.

Given the sequence $\mathbf{x} = [1.0, 0.5, -1.0]$ and an initial memory $\mathbf{h_0} = \mathbf{0}$, let us calculate the evolution of the hidden state.

The Temporal Walkthrough

Time Step $t = 1$: Processing the First Signal
The network receives $x_1 = 1.0$. Since there is no prior history ($h_0 = 0$), the state is determined purely by the input:

$$h_1 = \tanh(0.6 \times 1.0 + 0.8 \times 0) = \tanh(0.6) \approx \mathbf{0.537}$$

Time Step $t = 2$: Merging Present and Past
Now $x_2 = 0.5$ arrives. The network retrieves its previous state $h_1 \approx 0.537$:

$$h_2 = \tanh(0.6{\times}0.5{+}0.8{\times}0.537) = \tanh(0.3{+}0.4296) = \tanh(0.7296) \approx \mathbf{0.623}$$

Notice how h_2 is larger than h_1; the positive signals are accumulating in the memory.

Time Step $t = 3$: Handling a Conflict

Finally, $x_3 = -1.0$ (a strong negative signal). It must compete with the existing positive memory $h_2 \approx 0.623$:

$$h_3 = \tanh(0.6\times(-1.0)+0.8\times 0.623) = \tanh(-0.6+0.4984) = \tanh(-0.1016) \approx \mathbf{-0.101}$$

The memory has shifted from positive to negative, but the "momentum" of the previous steps prevented it from dropping as low as -0.76 *(the* tanh *of* -1 *alone).*

Pedagogical Observation

The final state h_3 is a "summary" of the entire sequence. It is not just a reflection of x_3, but a weighted integration of the history $[x_1, x_2, x_3]$. This is the essence of temporal context.

6.8.1 Backpropagation Through Time (BPTT)

Training an RNN presents a unique challenge: the weights W_{xh} and W_{hh} are reused at every time step. To calculate how much to change a weight, we must determine its influence on the loss across the *entire* sequence. This is handled by **Backpropagation Through Time (BPTT)**.

The Computational Chain

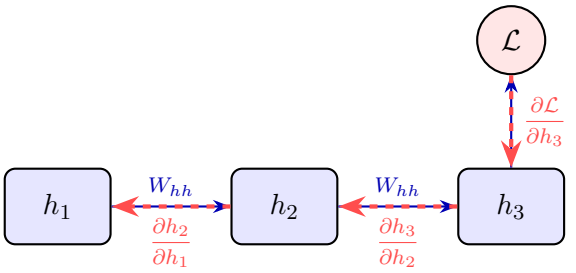


Figure 6.9: Backpropagation Through Time (BPTT): gradients from the loss at $t = 3$ propagate backward through time to update shared recurrent weights.

In BPTT, we compute the derivative of the loss with respect to the weights by summing the contributions from each time step. Because of

the chain rule, calculating the gradient at the start of a sequence requires multiplying together the gradients of all subsequent steps.

6.8.2 The Dual Crisis: Vanishing and Exploding Gradients

The chain-like structure of BPTT is its greatest weakness. Since $\frac{\partial \mathcal{L}}{\partial h_1}$ involves a product of many derivatives (e.g., $\dots \frac{\partial h_3}{\partial h_2} \times \frac{\partial h_2}{\partial h_1}$), the gradient is susceptible to exponential instability.

1. Vanishing Gradients: The Memory Fade

If the weights (W_{hh}) or the derivatives of the activation function are small (less than 1), the gradient shrinks as it travels back in time. By the time it reaches the first few steps of a long sequence, the signal is virtually zero.

- **The Result**: The network "forgets" long-range dependencies and only learns from the most recent inputs.

2. Exploding Gradients: The Numerical Shock

Conversely, if the weights are large, the gradient can grow exponentially.

- **The Result**: Massive weight updates that "shatter" the model, leading to `NaN` values and total failure of the training process.

[Image showing vanishing vs exploding gradient magnitude over time steps]

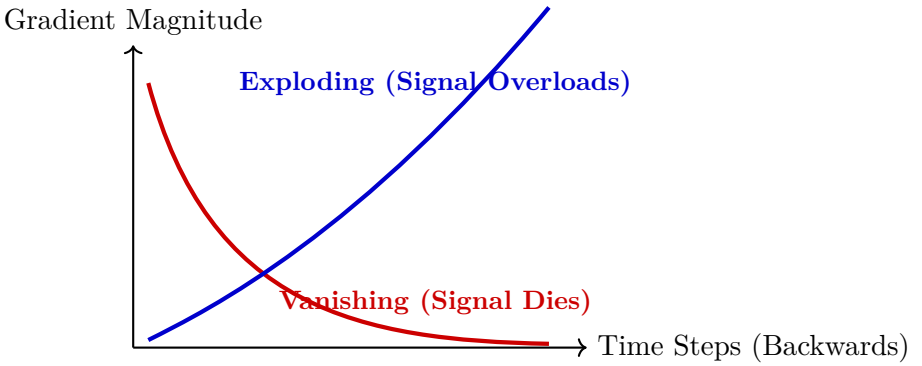


Figure 6.10: Visualizing the exponential decay or growth of training signals in standard RNNs.

6.8.3 Transitioning to Advanced Cells (LSTM/GRU)

The gradient instability of "Vanilla" RNNs makes them unsuitable for tasks requiring long-term memory, such as understanding a paragraph or predicting long-range weather patterns.

Consider the sequence: *"The **clouds** in the sky, which had been gray all morning, finally produced **rain**."* To connect "clouds" to "rain," the gradient must survive a dozen time steps.

To solve this, researchers introduced **Gating Mechanisms**:

- **LSTM (Long Short-Term Memory)**: Uses three gates to protect and maintain a "Cell State" (a dedicated long-term memory highway).
- **GRU (Gated Recurrent Unit)**: A simplified version of the LSTM that merges gates for efficiency.

The Path Forward

While the Vanilla RNN gave us the **concept** of memory, the LSTM and GRU gave us the **control** over that memory. In the next section, we will explore the "Constant Error Carousel" that allows LSTMs to bypass the vanishing gradient problem entirely.

6.8.4 Long Short-Term Memory (LSTM) Networks

Long Short-Term Memory (LSTM) networks were proposed to overcome the fundamental limitations of standard Recurrent Neural Networks (RNNs), particularly the **vanishing gradient problem**. Unlike conventional RNNs, LSTMs are capable of learning **long-range temporal dependencies** by introducing an explicit memory mechanism regulated by gating structures.

Why LSTMs Are Needed

In a standard RNN, the hidden state is updated at each time step using a single nonlinear transformation. As information is repeatedly propagated through time, it is continuously squashed by activation functions such as tanh or σ, which causes gradients to diminish rapidly during backpropagation.

LSTMs address this limitation by:

- Introducing a **cell state** C_t that flows through time with only minor linear interactions
- Employing **gating mechanisms** to selectively retain, update, or discard information

Core Idea of LSTM

An LSTM separates long-term memory storage (cell state C_t) from short-term memory usage (hidden state h_t), allowing important information to persist over long sequences.

Internal Structure of an LSTM Cell

Figure **??** illustrates the internal structure of a single LSTM cell, consistent with the mathematical formulation and signal flow shown in the accompanying diagram.

An LSTM cell consists of four main components:

1. **Forget gate** f_t, which controls how much of the previous cell state C_{t-1} is retained

2. **Input gate** i_t, which regulates how much new information is written into the cell

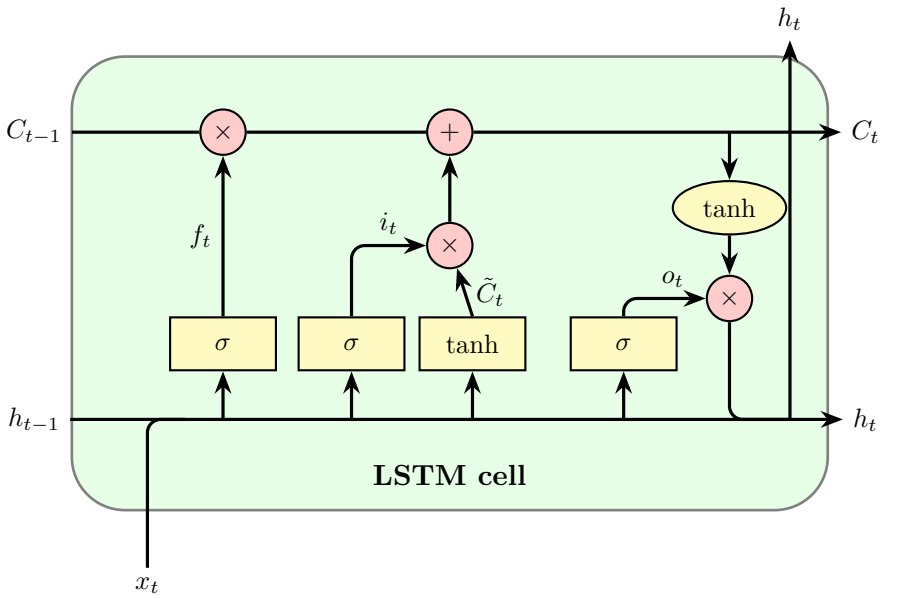
3. **Candidate cell state** $\tilde{C}_t$, which represents new memory content generated using a tanh activation

4. **Output gate** o_t, which determines how much of the updated cell state contributes to the hidden state

As depicted in the LSTM cell diagram, the forget gate modulates the previous cell state through a pointwise multiplication, while the input gate scales the candidate memory before it is added to the cell state. This results in the updated cell state C_t, which is then passed through a tanh activation and filtered by the output gate to produce the hidden state h_t.

This gated architecture allows the LSTM to preserve relevant information over long time horizons while discarding irrelevant or outdated signals in a controlled manner.

LSTM Cell Diagram



Forget Gate

The forget gate controls how much information from the previous cell state C_{t-1} is allowed to pass through the LSTM cell-state highway.

Forget Gate: Mathematical Formulation

As illustrated in the LSTM cell diagram, the forget gate is the first gating mechanism applied to the memory flow. It computes a gating vector that determines which components of the previous memory should be retained or erased. The forget gate is defined as:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Parameter Interpretation:

- $\sigma(\cdot)$ **(Sigmoid Activation)**: Produces values in the interval $(0, 1)$, enabling soft decisions about memory retention. Values close to 0 block information flow, while values close to 1 allow information to pass unchanged.
- f_t **(Forget Gate Vector)**: A gating vector with the same dimensionality as the cell state C_{t-1}. It performs element-wise filtering of the previous memory through the Hadamard product.

- W_f **(Weight Matrix)**: Learns how the previous hidden state h_{t-1} and current input x_t jointly influence the forgetting behavior.
- $[h_{t-1}, x_t]$ **(Concatenation)**: Represents the combined contextual and current input information provided to the gate.
- b_f **(Bias Vector)**: Allows the network to learn a default forgetting or remembering behavior.

Effect on the Cell State

In accordance with the multiplicative interaction shown in the diagram, the forget gate directly scales the previous cell state:

Forget Gate Action

$$\text{Retained Memory} = f_t \odot C_{t-1}$$

- $f_t^{(i)} \approx 1$: The i-th memory component is preserved.
- $f_t^{(i)} \approx 0$: The i-th memory component is erased.
- $0 < f_t^{(i)} < 1$: The memory component is partially attenuated.

The forget gate enables the LSTM to remove irrelevant historical information, preventing uncontrolled growth of the memory content.

Input Gate and Candidate Cell State

The input gate regulates the incorporation of new information into the cell state, as depicted by the lower gating pathway in the LSTM diagram.

Input Gate

The input gate determines how much newly computed information should be written to memory:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

The gating vector i_t scales the candidate memory before it is added to the cell state.

Candidate Cell State

The candidate cell state represents potential new memory content generated using a tanh activation:

$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

Cell State Update

As shown in the cell-state highway of the diagram, the updated memory is obtained through an additive interaction:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

Memory Update Mechanism

Old memory is selectively forgotten, while new, relevant information is selectively incorporated into the cell state.

6.8.5 Output Gate and Hidden State

The output gate determines which components of the updated cell state should influence the hidden state and be exposed to the next time step.

Output Gate

The output gate is defined as:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

Hidden State Computation

As illustrated in the upper-right portion of the LSTM cell diagram, the hidden state is computed by modulating the activated cell state with the output gate:

$$h_t = o_t \odot \tanh(C_t)$$

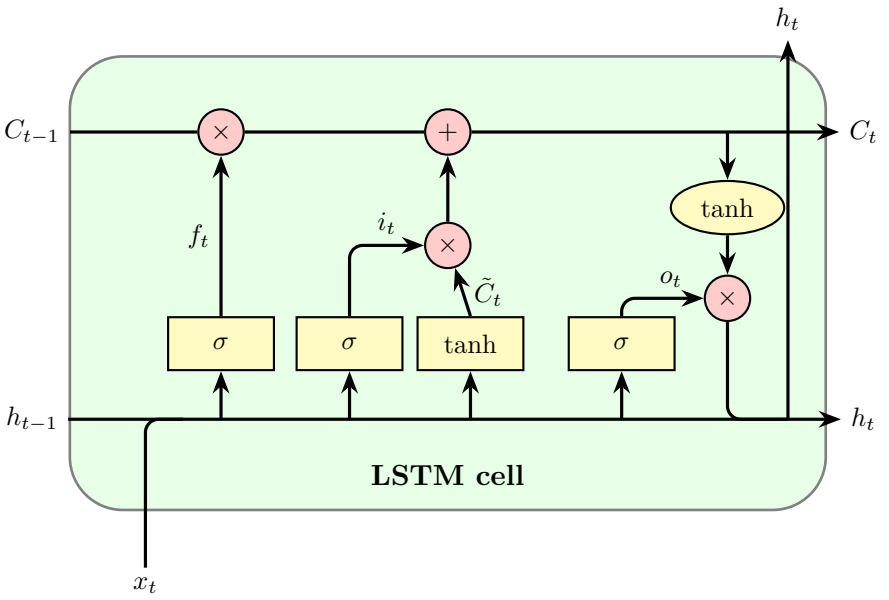
This final gating operation allows the LSTM to expose only the most relevant portions of its internal memory to subsequent layers or time steps.

- Cell state: long-term memory
- Hidden state: short-term working memory

LSTM Cell Equations

$$
\begin{aligned}
i_t &= \sigma\Big(x_t U^i + h_{t-1} W^i\Big) \\
f_t &= \sigma\Big(x_t U^f + h_{t-1} W^f\Big) \\
o_t &= \sigma(x_t U^o + h_{t-1} W^o) \\
\tilde{C}_t &= \tanh(x_t U^g + h_{t-1} W^g) \\
C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \\
h_t &= \tanh(C_t) \odot o_t
\end{aligned}
$$

Hidden State Extraction Diagram



Why LSTMs Solve the Vanishing Gradient Problem

The key innovation of LSTM lies in the **linear flow of the cell state**:

$$c_t = f_t \odot c_{t-1} + \dots$$

This additive structure allows gradients to propagate backward through many time steps with minimal decay.

Key Insight

LSTM cell states act like conveyor belts for gradients, preserving long-term information.

6.9 Gated Recurrent Unit (GRU)

The Gated Recurrent Unit (GRU) is a simplified variant of the LSTM that retains the ability to model long-term dependencies while reducing architectural complexity. GRUs were introduced to provide comparable performance with fewer parameters and faster training.

Unlike LSTMs, GRUs:

- Do not maintain a separate cell state
- Combine the forget and input gates into a single **update gate**
- Use fewer weight matrices

Design Philosophy

GRUs trade architectural complexity for computational efficiency while preserving learning capability.

6.9.1 Internal Structure of a GRU Cell

A GRU cell consists of two gates:

1. Update gate
2. Reset gate

GRU Cell Diagram

Here is the comprehensive LaTeX code using the TikZ library to recreate the provided LSTM cell diagram. This code utilizes standard neural network colors (green for container, yellow for layers, and pink/red for operations).

Mathematical Formulation of the GRU Cell

The Gated Recurrent Unit (GRU) modulates the flow of information through two primary gating mechanisms: the **Reset gate** (r_t) and the **Update gate** (z_t). These gates, along with the candidate hidden state, are defined as follows:

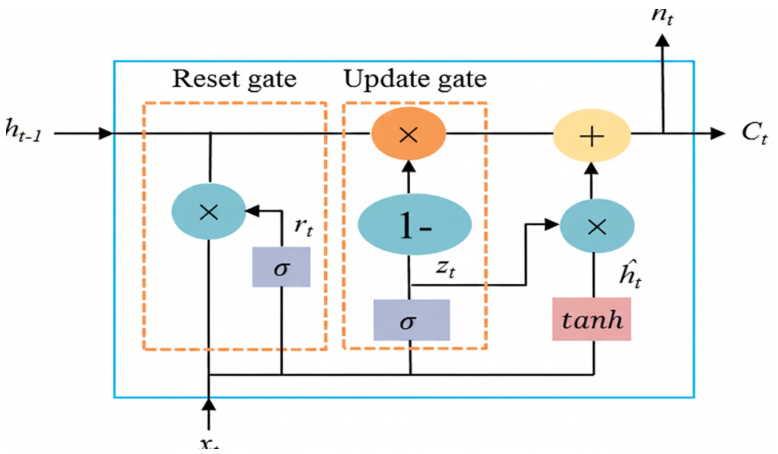


Figure 6.11: GRU cell showing reset gate, update gate, candidate hidden state, and final hidden state computation.

1. Gating Mechanisms

Both gates utilize a sigmoid activation function σ to squash the output between 0 and 1, determining the degree to which information is passed.

- **Reset Gate (r_t):** Controls how much of the previous hidden state h_{t-1} is ignored when calculating the new candidate state.

$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1}) \tag{6.1}$$

- **Update Gate (z_t):** Dictates the proportion of the previous hidden state that should be carried forward to the current state.

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1}) \tag{6.2}$$

2. Candidate Hidden State ($\hat{h}_t$)

The candidate hidden state (labeled as h'_t or $\hat{h}_t$ in the diagram) uses the reset gate to filter the influence of the past state on the current input x_t.

$$\hat{h}_t = \tanh(Wx_t + r_t \odot (Uh_{t-1})) \tag{6.3}$$

3. Final Hidden State (h_t)

The final hidden state h_t is a weighted sum of the previous hidden state h_{t-1} and the new candidate state $\hat{h}_t$. This is represented by the $1 - z_t$ logic in the diagram.

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t \tag{6.4}$$

- x_t: Input vector at time step t.
- h_{t-1}: Hidden state output from the previous time step.
- $W^{(r)}, W^{(z)}, W$: Weight matrices for inputs.
- $U^{(r)}, U^{(z)}, U$: Weight matrices for hidden states.
- σ: Sigmoid activation function.
- $\odot$: Element-wise (Hadamard) multiplication.

6.10 LSTM vs GRU: A Comparative Analysis

Table 6.1: Architectural Comparison: LSTM vs. GRU

Feature	LSTM	GRU
Memory Units	Separate **Cell State** (c_t) and **Hidden State** (h_t).	Unified **Hidden State** (h_t) only
Gating Logic	Three gates: **Forget**, **Input**, and **Output**.	Two gates: **Update** and **Reset**.
Parameters	**Higher count**; more weights due to additional gates.	**Lower count**; computationally ficient.
Training Speed	Generally **slower** due to complex internal routing.	Generally **faster** due to fewer op ations.
Sequence Modeling	Excellent for **very long-range dependencies**.	Highly effective; often matc LSTM on most tasks.
Architectural Complexity	**More complex**; separates long-term memory from output.	**Simpler design**; hidden state a as both memory and output.

6.11 RNN vs LSTM vs GRU

Table 6.2: Comparative Analysis: Recurrent Neural Network Architectures

Model	Core Strengths	Key Technical Limitations
RNN	Simple architecture; computationally inexpensive for short sequences.	**Vanishing/Exploding Gradients:** Difficulty capturing dependencies over more than 10–20 steps.
LSTM	**Constant Error Carousel:** Effectively preserves gradients over long time steps using its cell state.	**High Complexity:** More parameters to train (4× more than standard RNN) and higher memory footprint.
GRU	**Streamlined Logic:** Merges input and forget gates into an update gate; faster to train than LSTM.	**Reduced Expressivity:** Lack of a separate output gate can slightly limit performance on highly complex datasets.

Chapter 7

Reinforcement Learning

7.1 Reinforcement Learning

The RL Paradigm

In RL, the agent takes **Actions**, moves into new **States**, and receives **Rewards**. The goal of the agent is not to minimize error, but to **maximize the cumulative reward** over time.

Unlike Supervised Learning, which learns from a fixed dataset of labeled examples, **Reinforcement Learning (RL)** is a computational approach to learning through interaction. The learner and decision-maker is called the **Agent**, and the thing it interacts with is called the **Environment**.

Core Idea of Reinforcement Learning

An agent observes the current state of the environment, takes an action, receives a reward, and gradually learns which actions lead to the best long-term outcomes.

Reinforcement Learning (RL) is a learning framework in which an intelligent **agent** learns how to make decisions by **interacting with an environment**. Unlike supervised learning, there are no labeled examples. Instead, the agent discovers good behavior through **trial and error** by receiving feedback in the form of **rewards**.

Every reinforcement learning problem is built upon five essential components:

- **Agent**: The learner or decision-maker.
- **Environment**: The external system the agent interacts with.

- **State (s_t)**: A representation of the current situation.
- **Action (a_t)**: A choice made by the agent.
- **Reward (r_{t+1})**: A numerical signal indicating the quality of the action.

These components interact repeatedly, forming the reinforcement learning loop.

Example: A Robot Moving Between Rooms

Consider a very simple environment consisting of **two rooms**. A robot must learn how to move between them.

- **States**: $\{\text{Room 1}, \text{Room 2}\}$
- **Actions**: $\{\text{Move Left}, \text{Move Right}\}$
- **Goal**: Reach Room 2

At time step t, the robot starts in **Room 1**. If it selects the correct action, it receives a positive reward.

Time	State (s_t)	Action (a_t)	Reward (r_{t+1})
t	Room 1	Move Right	$+1$
$t+1$	Room 2		

Interpretation

The robot receives a positive reward because it selected an action that moved it closer to the goal. This encourages the agent to repeat the same decision in similar situations.

The Reinforcement Learning Cycle

The learning process follows a continuous loop:

1. The agent observes the current state s_t
2. The agent selects an action a_t
3. The environment transitions to a new state s_{t+1}
4. The agent receives a reward r_{t+1}

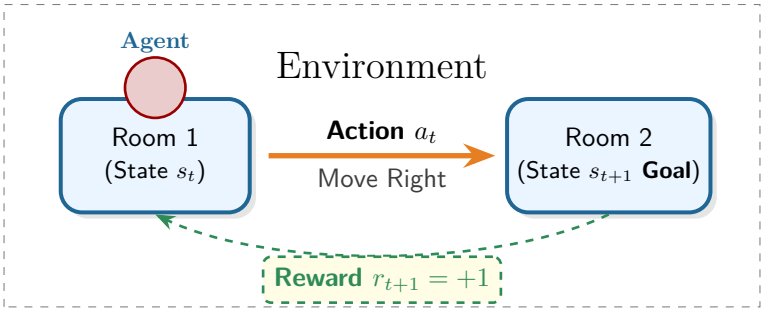


Figure 7.1: Visualizing the Transition: The agent executes a move that changes the state of the environment and triggers a scalar feedback signal (reward).

Over many interactions, actions that yield higher rewards become more likely.

Important point

Reinforcement Learning mimics how humans and animals learn: good outcomes strengthen behavior, while poor outcomes weaken it.

7.2 The Markov Decision Process (MDP) Framework

To solve an RL problem, we must formalize it as a **Markov Decision Process**. This framework assumes that the "future is independent of the past, given the present." This is known as the **Markov Property**.

Components of an MDP

A formal MDP is defined by the following tuple:

- **State Space (S):** A set of all possible configurations the agent can inhabit.
- **Action Space (A):** All possible moves available to the agent.
- **Transition Probability (P):** $P(s'|s, a)$ is the probability of reaching state s' given action a in state s.
- **Reward Function (R):** $R(s, a)$ is the numerical feedback received after an action.

- **Discount Factor (γ):** A value in $[0, 1]$ that represents the preference for immediate rewards over future rewards.

Example: Markov Decision Process (MDP)

To clearly understand a **Markov Decision Process (MDP)**, let us study a small Example where all components=states, actions, rewards, and transitions=are explicitly defined.

Consider a robot moving between **three locations** arranged in a straight line.

- **State Space:**
$$S = \{S_1, S_2, S_3\}$$
where:
  - S_1 = Left Room
  - S_2 = Middle Room
  - S_3 = Right Room (Goal)
- **Action Space:**
$$A = \{\text{Left}, \text{Right}\}$$
- **Reward Function:**
$$R(s, a) = \begin{cases} +10 & \text{if the agent reaches } S_3 \\ -1 & \text{for every other move} \end{cases}$$
- **Discount Factor:**
$$\gamma = 0.9$$

The objective of the agent is to reach the goal state S_3 while maximizing cumulative reward.

Transition Probabilities

The environment is deterministic, meaning the next state is fully determined by the current state and action.

Current State	Action	Next State
S_1	Right	S_2
S_2	Right	S_3
S_2	Left	S_1
S_3	Any	S_3 (Terminal)

Thus,
$$P(s'|s, a) = 1 \quad \text{for the transitions above}$$

MDP Interaction Example

Suppose the agent starts in state S_1 and follows the policy:

$$\pi(s) = \text{Right for all } s$$

Time	State S_t	Action A_t	Reward R_{t+1}
t	S_1	Right	-1
$t+1$	S_2	Right	$+10$
$t+2$	S_3		

Return Calculation

The return starting from S_1 is:

$$G_0 = -1 + 0.9 \times 10 = 8$$

Interpretation

Even though the agent receives a negative reward initially, the discounted future reward dominates, making the overall decision beneficial.

Visual Representation of the MDP

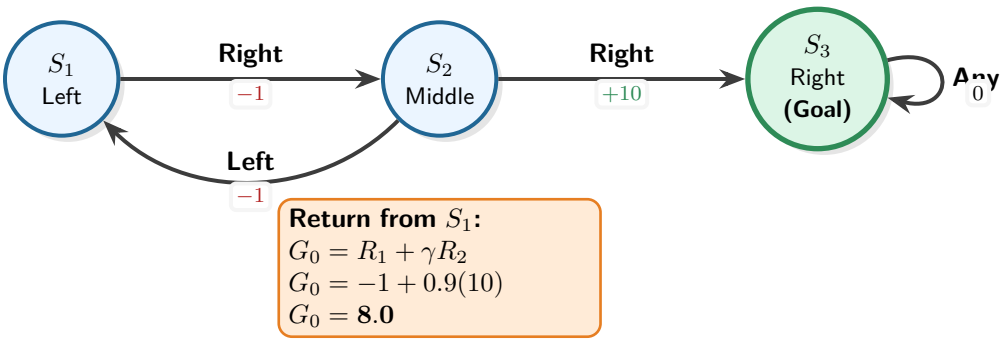


Figure 7.2: Markov Decision Process (MDP) Transition Graph: A deterministic environment where the agent learns to navigate toward the high-reward terminal state S_3.

This example satisfies all MDP properties:

- **Markov Property:** The next state depends only on the current state and action.

- **Sequential Decisions:** Actions influence future rewards.
- **Quantified Rewards:** Each action produces numerical feedback.
- **Optimization Goal:** Maximize expected return.

Important Point

An MDP provides the mathematical foundation that allows reinforcement learning algorithms to reason about the future before making decisions.

7.2.1 The Goal: Maximizing Expected Return

The agent's objective is to maximize the **Return** (G_t), which is the total discounted sum of rewards from time step t onwards.

The Discounted Return Formula

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

- If $\gamma = 0$, the agent is *myopic* (cares only about the next step).
- If $\gamma = 1$, the agent is *farsighted* (values future rewards as much as immediate ones).

7.2.2 Value Functions and Policies

A **Policy** (π) is the strategy that the agent uses to determine the next action based on the current state.

- **Deterministic Policy:** $a = \pi(s)$
- **Stochastic Policy:** $\pi(a|s) = P(A_t = a|S_t = s)$

State-Value Function $V^\pi(s)$

This function estimates "how good" it is for the agent to be in a given state under policy π:

$$V^\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

Action-Value Function $Q^\pi(s, a)$

Commonly known as the **Q-Function**, this estimates the value of taking a specific action a in state s:

$$Q^\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$

7.2.3 State-Value Function

The **State-Value Function** answers a very important question:

> *"If the agent starts in a particular state and follows a given policy, how much total reward can it expect in the future?"*

Mathematically, the value function is defined as:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t \mid S_t = s]$$

We reuse the same robot navigation MDP:

- States: $S = \{S_1, S_2, S_3\}$
- Actions: $\{\text{Left}, \text{Right}\}$
- Rewards:
  - $+10$ for reaching S_3
  - -1 for every other move
- Discount factor: $\gamma = 0.9$
- S_3 is a terminal state

Policy Definition

Assume the agent follows the deterministic policy:

$$\pi(s) = \text{Always move Right}$$

This means:

$$\pi(\text{Right} \mid s) = 1$$

Computing the Value of Each State

Value of Terminal State S_3 Since S_3 is terminal and no future rewards are received:

$$V^{\pi}(S_3) = 0$$

Value of State S_2 From S_2, the agent moves Right to S_3 and receives a reward of $+10$.

$$V^{\pi}(S_2) = 10 + \gamma V^{\pi}(S_3)$$

$$V^{\pi}(S_2) = 10 + 0.9 \times 0 = 10$$

Value of State S_1 From S_1, the agent moves Right to S_2 and receives a reward of -1.

$$V^{\pi}(S_1) = -1 + \gamma V^{\pi}(S_2)$$

$$V^{\pi}(S_1) = -1 + 0.9 \times 10 = 8$$

Final Value Function

$$V^{\pi}(S_1) = 8, \quad V^{\pi}(S_2) = 10, \quad V^{\pi}(S_3) = 0$$

Key Observation

States closer to the goal have higher value because they lead to larger future rewards with less penalty.

Bellman Equation Verification

The Bellman Expectation Equation is:

$$V^{\pi}(s) = r(s, a) + \gamma V^{\pi}(s')$$

Each computed value satisfies this recursive relationship, proving that the values are **Bellman-consistent**.

Visual Interpretation

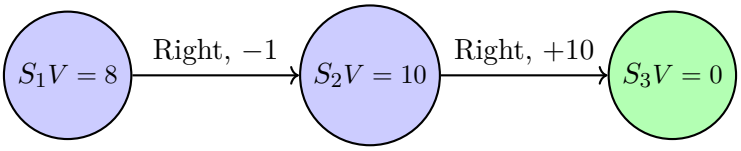


Figure 7.3: State values increase as the agent moves closer to the goal.

Why the Value Function Matters

The significance of the value function lies in its ability to predict **long-term benefit** rather than focusing solely on immediate rewards. By evaluating states based on their future potential, policies are effectively improved by strategically choosing actions that lead to **higher-value states**. As a result, value functions serve as the essential mathematical foundation for the most critical algorithms in Reinforcement Learning, including *Policy Iteration*, *Value Iteration*, *Q-Learning*, and modern *Deep RL* architectures.

7.2.4 Action-Value Function (Q-Function)

While the **Value Function** tells us how good a *state* is, it does not tell us *which action* caused that value. This limitation is resolved by the **Action-Value Function**, commonly called the **Q-Function**.

> *"How good is it to take a specific action in a given state and then follow a policy?"*

Mathematically, the Q-function is defined as:

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}\left[G_t \mid S_t = s, A_t = a\right]$$

We again use the same simple navigation problem:

- States: $S = \{S_1, S_2, S_3\}$
- Actions: $\{\text{Left}, \text{Right}\}$
- Rewards:
  - $+10$ for reaching S_3
  - -1 for every non-terminal move
- Discount factor: $\gamma = 0.9$
- S_3 is a terminal state

Policy Assumption

Assume the agent follows the policy:

$$\pi(s) = \text{Always move Right}$$

Key Distinction

- $V^\pi(s)$: Value of being in state s
- $Q^\pi(s, a)$: Value of taking action a in state s

Step-by-Step Q-Value Calculations

1. Q-Value of Terminal State Since S_3 is terminal:

$$Q^\pi(S_3, a) = 0 \quad \forall a$$

2. Q-Value at State S_2 From S_2:

- **Action: Right** $\rightarrow S_3$ with reward $+10$
- **Action: Left** $\rightarrow S_1$ with reward -1

$$Q^\pi(S_2, \text{Right}) = 10 + \gamma V^\pi(S_3) = 10$$

$$Q^\pi(S_2, \text{Left}) = -1 + \gamma V^\pi(S_1) = -1 + 0.9 \times 8 = 6.2$$

3. Q-Value at State S_1 From S_1:

- **Action: Right** $\rightarrow S_2$ with reward -1
- **Action: Left** $\rightarrow S_1$ (self-loop) with reward -1

$$Q^\pi(S_1, \text{Right}) = -1 + \gamma V^\pi(S_2) = -1 + 0.9 \times 10 = 8$$

$$Q^\pi(S_1, \text{Left}) = -1 + \gamma V^\pi(S_1) = -1 + 0.9 \times 8 = 6.2$$

Final Q-Table

Interpretation

The optimal action in each state is the one with the **highest Q-value**.

State	Left	Right
S_1	6.2	**8.0**
S_2	6.2	**10.0**
S_3	0	0

Table 7.1: Action-Value (Q) Table

Relation to the Bellman Equation

The Bellman Expectation Equation for Q-values is:

$$Q^{\pi}(s,a) = r + \gamma \sum_{s'} p(s'|s,a) V^{\pi}(s')$$

Each value in the Q-table satisfies this recursive definition.

Visual Interpretation

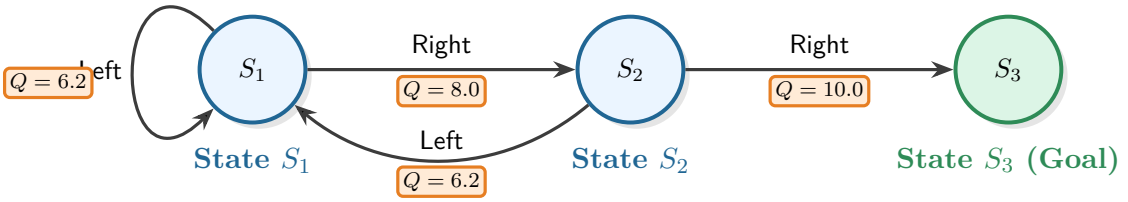


Figure 7.4: Visualizing the Q-Table on the Graph: Each arrow represents a decision a from state s, and the orange boxes represent the expected long-term value $Q(s,a)$.

Important Point

An agent doesn't need to know the physics of the environment if it has a complete Q-table. It simply looks at its current state and picks the action with the **maximum orange box value**.

7.3 The ϵ-Greedy Strategy

Once an agent has estimated the **Q-values**, it still faces a critical decision:

Should I exploit what I already know, or explore to discover something better?

This dilemma is known as the **Exploration–Exploitation Trade-off**. The most widely used solution is the **ϵ-Greedy Strategy**.

Visual Interpretation: Decision Branching

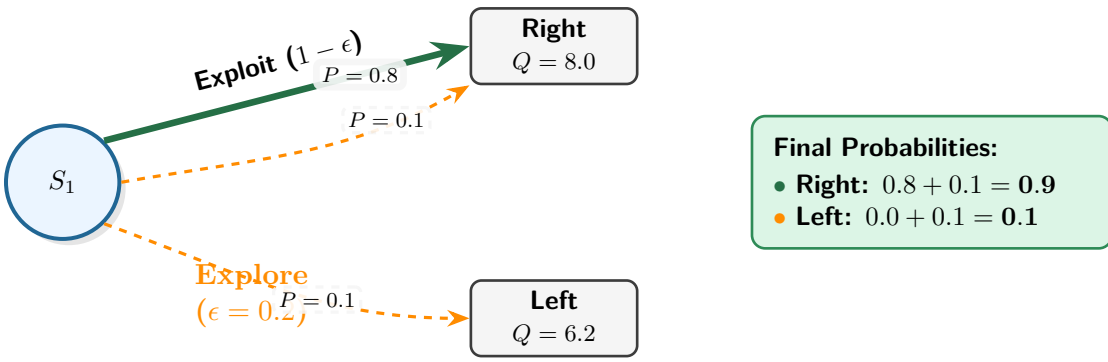


Figure 7.5: Visualizing the ϵ-greedy mechanism. Thick lines indicate high-probability greedy choices, while dashed lines represent the stochastic exploration of all available actions.

Key Implementation Details

Why Two Arrows to "Right"?

Students often ask why there are two arrows pointing to the **Right** action.

1. The agent chooses Right because it is the **Best Action** (Exploitation).
2. The agent chooses Right by **Pure Chance** while exploring (Exploration).

This is why the final probability is 0.9 rather than just 0.8.

Important Point

The value function evaluates *states*, but the Q-function evaluates *decisions*. Reinforcement Learning becomes actionable only when Q-values are known.

Important Point

The value function teaches an agent to think ahead. A state with a small immediate penalty can still be valuable if it leads to large future rewards.

7.4 The Exploration-Exploitation Trade-off

A fundamental challenge in RL is balancing **Exploration** (trying new things to find better rewards) and **Exploitation** (using existing knowledge to get known rewards).

ϵ-Greedy Strategy

To solve this, we often use the ϵ-greedy algorithm:

- With probability ϵ, choose a **random** action (Explore).
- With probability $1-\epsilon$, choose the **best** known action (Exploit).

7.5 The Bellman Equations: The Core of RL

The fundamental challenge in Reinforcement Learning is that an agent must make a decision now to maximize rewards that might not be received until much later. The **Bellman Equations** solve this by expressing the value of a state recursively.

The central idea of Reinforcement Learning is that the value of a decision today depends on the value of future decisions. The **Bellman Equations** formalize this idea using recursion.

> *"The value of a state equals the immediate reward plus the discounted value of the next state."*

Bellman Expectation Equation : For a given policy π, the Bellman Expectation Equation (Policy Evaluation) defines the state-value function as:

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma V^{\pi}(s') \right]$$

Interpretation

The value of a state is the **expected immediate reward** plus the **discounted value of successor states**, assuming the agent follows policy π.

Consider a simple MDP:

- States: $S = \{S_1, S_2, S_3\}$, where S_3 is terminal
- Actions: Left, Right
- Rewards:
  - $+10$ for reaching S_3
  - -1 for each non-terminal move
- Discount factor: $\gamma = 0.9$
- Policy π: Always move Right

Value of S_2: Moving Right from S_2 leads to S_3:

$$V^{\pi}(S_2) = 10 + 0.9 \times 0 = 10$$

Value of S_1: Moving Right from S_1 leads to S_2:

$$V^{\pi}(S_1) = -1 + 0.9 \times 10 = 8$$

TheBellman Optimality Equation: removes the dependence on a fixed policy and instead assumes the agent always chooses the *best* possible action:

$$V^*(s) = \max_{a \in A} \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma V^*(s') \right]$$

Key Idea

Instead of averaging over actions, the agent selects the action that yields the **maximum long-term reward**.

Example: Bellman Optimality

At state S_1:

$$Q(S_1, \text{Right}) = -1 + 0.9 \times 10 = 8$$
$$Q(S_1, \text{Left}) = -1 + 0.9 \times 8 = 6.2$$

$$V^*(S_1) = \max(8,\ 6.2) = 8$$

Thus, the optimal action in S_1 is **Right**.

Bellman Equation for the Q-Function

The Bellman Optimality Equation for action-values is:

$$Q^*(s, a) = \sum_{s', r} p(s', r \mid s, a) \left[r + \gamma \max_{a'} Q^*(s', a') \right]$$

This equation is the foundation of:

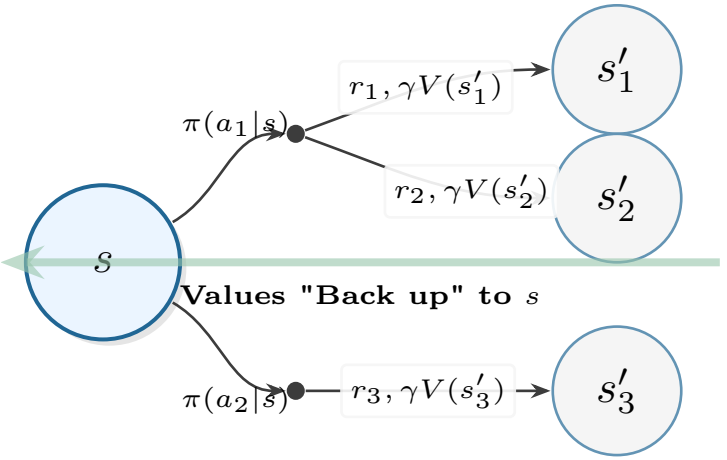
- Q-Learning
- SARSA
- Deep Q-Networks (DQN)

Visual Interpretation

- Bellman equations decompose long-term rewards into **local decisions**
- They enable recursive computation of values
- Every major RL algorithm is a practical approximation of a Bellman equation

Bellman Expectation Equation

This equation relates the value of a state to the values of its possible successor states under a specific policy π.



The Bellman Logic:
The value of s is the average of rewards (r) and future values ($V(s')$) weighted by probabilities.

Figure 7.6: Enlarged Bellman Backup Diagram illustrating the recursive propagation of value information.

Bellman Expectation Equation

The value of state s is the expected reward plus the discounted value of the next state:

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S, r \in R} p(s', r|s, a) \left[r + \gamma V^{\pi}(s') \right]$$

Bellman Optimality Equation

To find the best possible behavior, we don't look at the average of actions, but the **maximum**. This equation describes the value of a state under the **Optimal Policy** π^*.

Bellman Optimality Equation

$$V^*(s) = \max_{a \in A} \sum_{s',r} p(s', r|s, a) \left[r + \gamma V^*(s') \right]$$

7.6 Dynamic Programming (DP)

Dynamic Programming refers to a collection of algorithms that can be used to compute optimal policies given a perfect **model** of the environment as a Markov Decision Process (MDP). In DP, we assume the agent knows the transition probabilities $p(s', r|s, a)$—effectively having a "god's eye view" of the environment.

The Principle of Optimality

DP breaks down the problem of finding an optimal policy into two repeatable sub-problems:

1. **Policy Evaluation:** Computing the value function V_π for an arbitrary policy.
2. **Policy Improvement:** Making the policy better with respect to the current value function.

The Bellman Expectation Equation

The core of DP is the **full backup**. Unlike TD learning, which samples one transition, DP considers every possible next state and reward, weighted by their probability:

$$V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) [r + \gamma V_k(s')]$$

Key Algorithms: Policy and Value Iteration

Policy Iteration This algorithm alternates between full evaluation and improvement. It converges to the optimal policy π^* in a finite number of steps by repeatedly "tightening" the relationship between the values and the actions.

$$\pi_0 \xrightarrow{E} V_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V_{\pi_1} \xrightarrow{I} \dots \xrightarrow{I} \pi^* \xrightarrow{E} V^*$$

Value Iteration Value iteration simplifies this process by combining evaluation and improvement into a single update. It essentially applies the **Bellman Optimality Equation** as an update rule:

$$V_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V_k(s')]$$

Bootstrapping in DP

Like TD Learning, DP uses **bootstrapping**—it updates its estimate of $V(s)$ based on its estimates of $V(s')$. However, DP is unique because it is **Model-Based**: it uses the environment's true transition dynamics rather than relying on sampled experience.

Example: Gridworld Planning

Consider a 3×3 grid where an agent knows that moving *Right* from S_1 has a 100% probability of reaching S_2 with a reward of -1.

$$p(S_2, -1|S_1, \text{Right}) = 1.0$$

In DP, the value update for S_1 doesn't wait for the agent to move. It looks into the model:

$$V_{new}(S_1) = \underbrace{1.0}_{\text{Prob}} \times [\underbrace{-1}_{\text{Reward}} + \gamma V_{old}(S_2)]$$

The DP Limitation

While DP is mathematically elegant and provides exact solutions, it suffers from the **Curse of Dimensionality**. As the number of states increases, the computational cost of performing a "full sweep" over the entire state space becomes prohibitive.

7.7 Temporal Difference (TD) Learning

TD Learning is a sophisticated blend of Monte Carlo (MC) and Dynamic Programming (DP). It learns directly from raw experience without a model of the environment (like MC), but it updates estimates based in part on other learned estimates, without waiting for a final outcome (like DP). This process is known as **bootstrapping**.

The TD(0) Update Rule

Instead of waiting for the full return G_t, we update the value of a state using the immediate reward and the estimated value of the subsequent state:

$$V(S_t) \leftarrow V(S_t) + \alpha \underbrace{[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]}_{\text{TD Error}}$$

Model-Free Control: From States to Actions

To find the optimal policy in a setting where the environment's dynamics are unknown, we transition from state-values $V(s)$ to **action-values** $Q(s, a)$. This allows the agent to make decisions directly from the learned values. Two primary algorithms dominate this space: **SARSA** and **Q-Learning**.

> *SARSA learns from what it actually does (On-Policy).*
> *Q-Learning learns from what it could have done optimally (Off-Policy).*

Update Equations: On-Policy vs Off-Policy

SARSA (On-Policy) SARSA updates the Q-value based on the specific action A_{t+1} actually taken by the agent following its current ϵ-greedy policy:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [\underbrace{R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})}_{\text{Target}} - Q(S_t, A_t)]$$

Q-Learning (Off-Policy) Q-Learning ignores the actual next action and instead bootstraps using the **best possible** action available in the next state:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [\underbrace{R_{t+1} + \gamma \max_a Q(S_{t+1}, a)}_{\text{Target}} - Q(S_t, A_t)]$$

The Behavioral Difference

- **SARSA**: Incorporates the agent's exploration into the value update.
- **Q-Learning**: Directly estimates the value of the optimal policy, regardless of exploration.

Comparative Example

Consider a transition $(S_1, \text{Right}, -1, S_2)$ with $\alpha = 0.1, \gamma = 0.9$. Current values: $Q(S_1, \text{Right}) = 8$; $Q(S_2, \text{Left}) = 6.2$, $Q(S_2, \text{Right}) = 10$.

1. SARSA Update If the agent explores and chooses $A_{t+1} = \text{Left}$:

$$\text{TD Target} = -1 + 0.9(6.2) = 4.58$$

$$Q_{\text{new}}(S_1, \text{Right}) = 8 + 0.1(4.58 - 8) = \mathbf{7.66}$$

2. Q-Learning Update Q-Learning uses $\max_a Q(S_2, a) = 10$ regardless of the chosen action:

$$\text{TD Target} = -1 + 0.9(10) = 8$$

$$Q_{\text{new}}(S_1, \text{Right}) = 8 + 0.1(8 - 8) = \mathbf{8.00}$$

Observation

SARSA **penalizes risky exploration** (the update dropped because the agent took a sub-optimal path), while Q-Learning **assumes optimal future behavior** and maintains its high valuation.

Visual Comparison

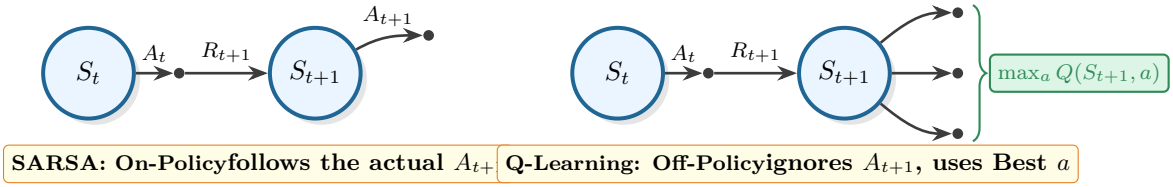


Figure 7.7: The visual difference in backups: SARSA follows a specific trajectory (the actual A_{t+1} taken), whereas Q-Learning "looks ahead" and updates based on the most valuable path available at S_{t+1}.

7.8 Monte Carlo (MC) Methods

If Dynamic Programming is "planning with a map," **Monte Carlo (MC)** is "learning from hindsight." Imagine playing a game where you don't know the rules. You play until you win or lose, and only then do you look back to see which moves were responsible for your final score.

The "Wait-and-See" Approach

Monte Carlo methods do not update their estimates after every step. Instead, they wait for a **complete episode** to finish. The agent learns by averaging the actual total returns it received.

Unlike other methods, MC does not "guess based on a guess." It uses the **Actual Return** (G_t). We update the Value of a state $V(S_t)$ by moving it slightly toward the total reward we actually saw:

$$V(S_t) \leftarrow V(S_t) + \alpha \underbrace{[G_t - V(S_t)]}_{\text{Prediction Error}}$$

Where G_t is the sum of all rewards from time t until the end of the game.

Example:

Let's look at a simple example to see the math in action.

1. The Setup An agent starts at State A. Currently, the agent thinks State A is worth nothing: $V(A) = 0$. We set the learning rate $\alpha = 0.1$.

2. The Episode The agent takes three steps to finish the game:

1. Start at $A \xrightarrow{step}$ receive Reward $R_1 = 2$
2. Move to $B \xrightarrow{step}$ receive Reward $R_2 = 1$
3. Move to $C \xrightarrow{step}$ receive Reward $R_3 = 10$ **(Goal Reached)**

3. The Hindsight Calculation The episode is over. The agent looks back and calculates the **Total Return**:

$$G = 2 + 1 + 10 = 13$$

4. The Value Update The agent updates its memory of State A:

$$V(A)_{new} = 0 + 0.1 \times (13 - 0) = \mathbf{1.3}$$

Observation: The agent didn't need to know the values of B or C to update A. It only needed the final score of 13.

Zero Bias: Because we use the actual G_t, our "target" is the truth. We aren't relying on other potentially incorrect estimates.

High Variance: This is the trade-off. Because episodes can be long and full of random choices, the final return G might be 100 in one game and -50 in the next.

Model-Free: MC is perfect when you don't know the "physics" of the world. You don't need to know the probability of moving from A to B; you just need to do it and record the reward.

The Main Limitation

Monte Carlo can **only** be used for episodic tasks. If a task never ends (like a continuous manufacturing process), there is no "Final Return," and the MC agent would wait forever to learn its first lesson!

Feature	MC	TD	DP
Learning	Episodic	Step-by-step	Planning
Bootstrapping	No	Yes	Yes
Model Needed	No	No	Yes
Bias / Variance	High Var / Zero Bias	Low Var / Some Bias	Low Var / Zero Bias

Comparison Summary:

Key Takeaway: Choosing Your Method

The choice between these three pillars depends entirely on your environment and your knowledge of the world:

- **Dynamic Programming (DP):** Use this if you have a **perfect model** (the "rules") of the environment. It is for planning and finding exact solutions offline.
- **Temporal Difference (TD):** Use this for **continuous tasks** (like keeping a drone level or a robot walking). It is the most efficient because it learns and corrects itself after every single step.
- **Monte Carlo (MC):** Use this for **episodic tasks with high randomness** (like card games or poker). It is more stable because it only cares about the actual final result, not intermediate guesses.

"DP plans with a map; TD learns as it walks; MC learns only after the journey is over."

7.9 Python Implementation: SARSA and Q-Learning

This section presents a complete and easy-to-follow Python implementation of two fundamental reinforcement learning algorithms: **SARSA** (on-policy) and **Q-Learning** (off-policy). A simple *Grid World* environment is used so that we can clearly understand how learning takes place step by step.

Environment Definition (Grid World)

```
1 import numpy as np
```

```python
import random

class GridWorld:
    def __init__(self):
        self.states = [(0,0), (0,1), (1,0), (1,1)]
        self.goal = (1,1)
        self.actions = ['up', 'down', 'left', 'right']

    def reset(self):
        return (0,0)

    def step(self, state, action):
        x, y = state

        if action == 'up':    x = max(x-1, 0)
        if action == 'down':  x = min(x+1, 1)
        if action == 'left':  y = max(y-1, 0)
        if action == 'right': y = min(y+1, 1)

        next_state = (x, y)

        if next_state == self.goal:
            return next_state, 10, True
        else:
            return next_state, -1, False
```

Listing 7.1: Simple Grid World Environment

ε-Greedy Action Selection

The ε-greedy policy balances **exploration** and **exploitation** by selecting a random action with probability ε and the best-known action otherwise.

```python
def epsilon_greedy(Q, state, actions, epsilon):
    if random.uniform(0,1) < epsilon:
        return random.choice(actions)   # Exploration
    else:
        return actions[np.argmax(Q[state])]  # Exploitation
```

Listing 7.2: Epsilon-Greedy Policy

SARSA Algorithm (On-Policy Learning)

SARSA updates its action-value function using the *actual action taken* under the current policy.

```python
def sarsa(env, episodes=500, alpha=0.1, gamma=0.9, epsilon=0.1):
    Q = {state: np.zeros(len(env.actions)) for state in env.states
    }

    for _ in range(episodes):
```

```
        state = env.reset()
        action = epsilon_greedy(Q, state, env.actions, epsilon)
        done = False

        while not done:
            next_state, reward, done = env.step(state, action)
            next_action = epsilon_greedy(Q, next_state, env.
    actions, epsilon)

            td_target = reward + gamma * Q[next_state][env.actions
    .index(next_action)]
            td_error = td_target - Q[state][env.actions.index(
    action)]
            Q[state][env.actions.index(action)] += alpha *
    td_error

            state, action = next_state, next_action

    return Q
```

Listing 7.3: SARSA Algorithm

Q-Learning Algorithm (Off-Policy Learning)

Q-Learning updates its values using the *maximum possible future reward*, independent of the agent's current policy.

```
def q_learning(env, episodes=500, alpha=0.1, gamma=0.9, epsilon
    =0.1):
    Q = {state: np.zeros(len(env.actions)) for state in env.states
    }

    for _ in range(episodes):
        state = env.reset()
        done = False

        while not done:
            action = epsilon_greedy(Q, state, env.actions, epsilon
    )
            next_state, reward, done = env.step(state, action)

            best_next = np.max(Q[next_state])
            td_target = reward + gamma * best_next
            td_error = td_target - Q[state][env.actions.index(
    action)]
            Q[state][env.actions.index(action)] += alpha *
    td_error

            state = next_state

    return Q
```

Listing 7.4: Q-Learning Algorithm

Execution and Comparison

```
env = GridWorld()

Q_sarsa = sarsa(env)
Q_qlearning = q_learning(env)

print("SARSA Q-values:")
for state in Q_sarsa:
    print(state, Q_sarsa[state])

print("\nQ-Learning Q-values:")
for state in Q_qlearning:
    print(state, Q_qlearning[state])
```

Listing 7.5: Running SARSA and Q-Learning

Key Learning Insight

- **SARSA** learns based on the actions the agent actually takes (safer, policy-aware).
- **Q-Learning** learns the optimal policy by always assuming the best future action.

Exam Tip

SARSA answers: *"What happens if I follow my current policy?"*
Q-Learning answers: *"What is the best possible action?"*