# Ultimate Cheatsheets for ML/DL

Mattia Bortolaso

October 2025

# Contents

# Chapter 1

# Introduction

This document serves as a comprehensive cheat sheet for studying Machine Learning and Deep Learning. It provides a structured summary of key algorithms, models, technologies, and optimization techniques, covering both foundational and state-of-the-art approaches.

The goal is to offer a concise yet rigorous reference for students, researchers, and practitioners seeking to review or deepen their understanding of modern artificial intelligence methodologies, including theoretical principles, implementation strategies, and code-level insights.

## 1.1 Artificial Intelligence (AI)

In this section we talk about Artificial Intelligence, in this chapter we talk in general about AI as we dive more deep later.

### 1.1.1 AI Introduction

**Artificial Intelligence (AI)** refers to the ability of machines or computers to mimic the way humans think and make decisions. AI enables machines to think like humans or to replicate certain functions of the human brain. In simple words, **AI is when we enable computers to think.**

Artificial Intelligence allows computers to understand, analyze data, and make decisions without human help or interaction. We use AI systems every day. A few common examples are *Siri*, *Alexa*, and *ChatGPT*.

Other categories where AI is used include:

- Virtual Assistants
- Social Media Algorithms
- Online Shopping Recommendations
- Predictive Text and Autocorrect
- Healthcare Diagnostics
- Language Translation Services
- Fraud Detection in Banking

AI enables computers to become intelligent with numbers and rules, performing calculations at incredible speed and with perfect accuracy. On the other hand, humans possess not only reasoning but also emotions, creativity, and the ability to adapt to complex and unpredictable situations.

### 1.1.2 Types of AI

Artificial Intelligence is divided based on two main categorization — based on capabilities and based on functionally of AI.

The following image illustrates these types of AI:

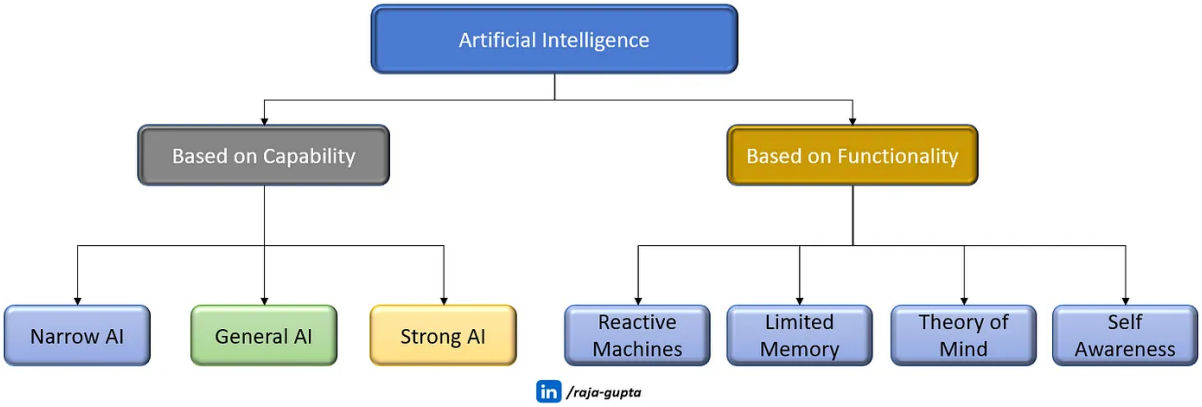


Figure 1.1: Types of AI

Now let's dive in the details of Narrow AI, General AI and Super AI.

- **Narrow AI:** Narrow AI, also known as Weak AI, refers to **artificial intelligence systems that are designed and trained for a specific task** or narrow set of tasks. Narrow AI is focused on performing a single task extremely well, but it cannot perform beyond its field or limitations.
- **General AI:** General AI, also known as Strong AI or artificial general intelligence (AGI), **can understand and learn any intellectual task that a human being can.**
- **Super AI:** Super AI represents a degree of intelligence in systems where **machines have the potential to exceed human intelligence**, outperforming humans in tasks and exhibiting cognitive abilities.

We can also divide types of AI based on functionality:

1. **Reactive Machines:** Reactive machines are AI **systems that have no memory**. These systems operate solely based on the present data, taking into account only the current situation. They can perform a narrowed range of pre-defined tasks.
2. **Limited Memory:** As the name indicates, Limited Memory AI can take informed and improved decisions by looking at its past experiences stored in a temporary memory.

This AI doesn't remember everything forever, but it uses its short-term memory to learn from the past and make better decisions for the future.

A **good example of Limited Memory AI is Self-driving cars**. The AI system in self-driving car utilizes recent past data to make real-time decisions. For instance, they employ sensors to recognize pedestrians, steep roads, traffic signals, and more, enhancing their ability to make safer driving choices. This proactive approach contributes to preventing potential accidents.

3. **Theory of Mind:** This is similar to Super AI — We should pray that we don't reach the state of AI, where **machines have their own consciousness and become self-aware**.

   Self-aware AI systems will be super intelligent, and will have their own consciousness, sentiments, and self-awareness. They will be smarter than human mind.

4. **Self-Aware AI:** This is similar to Super AI — We should pray that we don't reach the state of AI, where machines have their own consciousness and become self-aware.

   Self-aware AI systems will be super intelligent, and will have their own consciousness, sentiments, and self-awareness. They will be smarter than human mind.

   As shown in movie "I, Robot,", an AI system named VIKI becomes self-aware and starts making decisions to "protect" humanity in a controversial way.

   Similar to Theory of Mind, Self-aware AI also does not exist in reality. Many experts, for example Elon Musk and Stephen Hawkings have consistently warned us about the evolution of AI.

## 1.2 Machine Learning (ML):

In this chapter we are talking about Machine Learning, we first start explaining machine learning from a kids prospective and later we add more details.

### 1.2.1 ML Introduction:

Imagine we want to **enable the robot to identify several animals**.

To do so, we will show him pictures of various dogs, cats, bunnies and other animals and **label each picture with the name of the animal**. We train the robot to identify animals based on size, color, body shape, sound etc.

Once the training is completed, the robot will be able to identify these animals we trained him for.

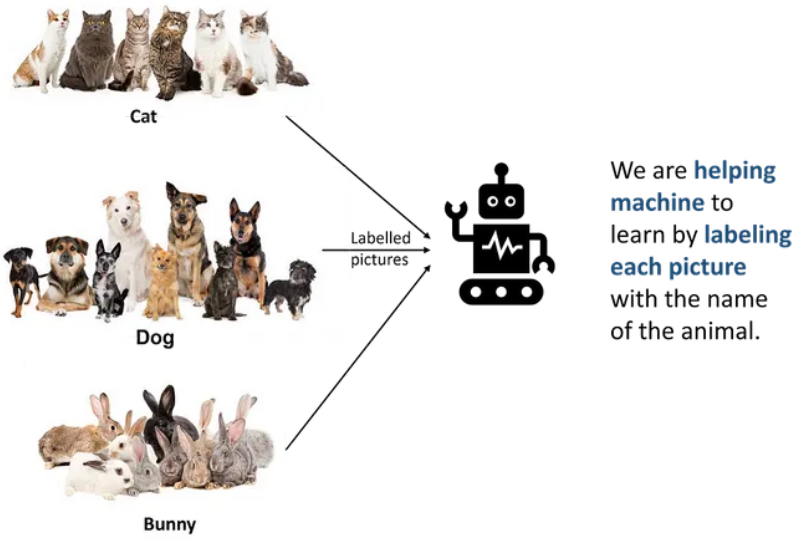


Figure 1.2: Machine Learning scheme

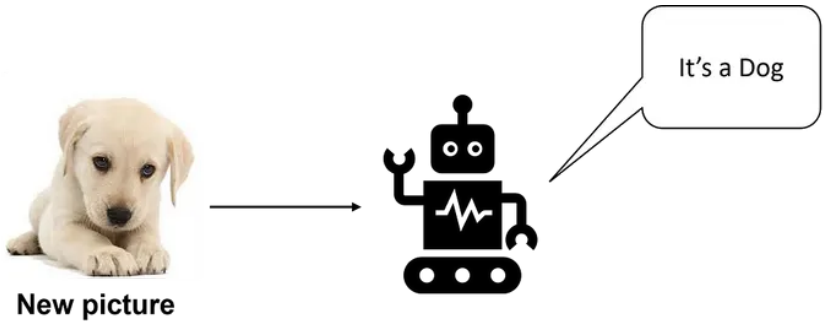


Figure 1.3: After training scheme

All dogs do not look alike. However, once robot has seen many pictures of dogs, **it can identify any dog even if it does not exactly look like a specific picture**. We need to show lots of pictures of dog to the robot. More pictures it sees, more efficient it will be.

In simple words Machine Learning is teaching a robot or a machine by giving a lot of examples to work with and learn.

### 1.2.2 Types of Machine Learning:

Machine Learning can be categorized into three main types:

1. Supervised Learning.
2. Unsupervised Learning.

3. Reinforcement Learning.

Each type serves different purposes and involves different approaches to learning from data. Let's have a close look into all these types.

### 1.2.3 Supervised Learning

**When we trained our robot by showing pictures of animals**, we labelled each picture with the name of the animal. So, **we acted as a teacher to him**. We first told him how does a dog or a cat look like and then only he was able to identify them.

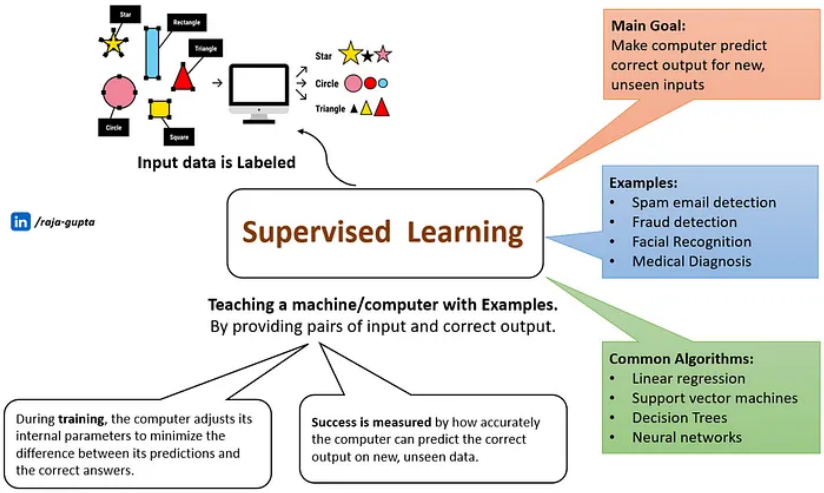


Figure 1.4: Supervised Learning summarized

Supervised Learning is widely used in this field: *Email Spamming Filtering, Image Classification, Facial Recognition, Financial Fraud Detection, Speech Recognition.*

### 1.2.4 Unsupervised Learning

Let's understand this from a kid's school example. When kids go to their class first day, they meet lots of classmates. At first all classmates are same to them. But with time, they themselves categorized them in different groups:

- They find some classmates very good and want to be friend with them.
- They find some rude or irritating and want to avoid them.
- They find some very good in sports and want to be in the same team as they are.
- and so on ...

**When kids categorized their classmates, nobody told them how to do that.** They did that without anyone's help. — This is how unsupervised learning works.

Let's take a proper machine learning example. Imagine we showed lots of pictures of dogs, cats, bunnies etc. **without any label** to our robot and told him — *"I'm not going to tell you which one is which. Go explore and figure it out".*

The robot starts to look at these animals, noticing things such as their fur, size, and how they move. It doesn't know their names yet, but it's trying to find patterns and differences on its own.

After exploring, the robot might notice that:

- Some animals have long ears (bunnies)
- Some animals have soft fur and tail (cats)
- Some animals have wagging tails (dogs)

In the end, the robot **might not know the names of the animals**, but it can say that **"These animals are similar in some ways, and those are different in other ways."** — This is Unsupervised Learning.

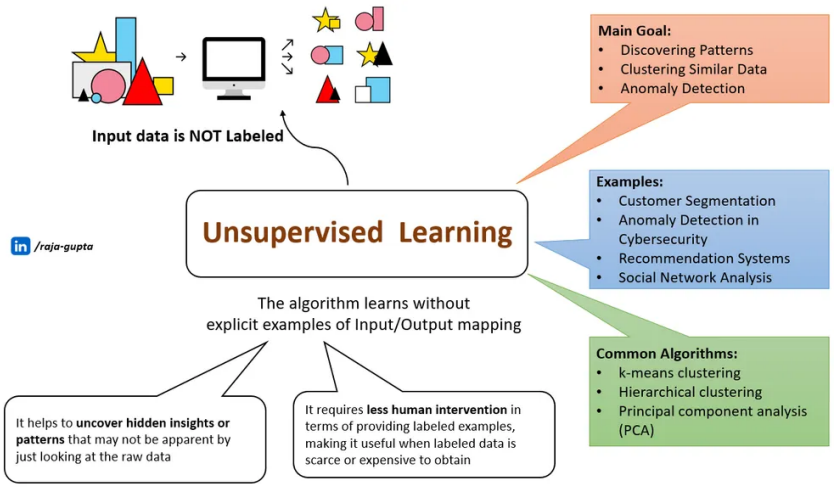


Figure 1.5: Unsupervised Learning summarized

Unsupervised Learning is widely used in this field: *Clustering Customer Segmentation, Anomaly Detection in Cybersecurity, Recommendation System.*

### 1.2.5 Reinforcement Learning

Imagine teaching a dog a new trick — you **reward it with a treat when it does the trick correctly** and give no treat when it doesn't. Over time, the dog **learns to perform the trick to get more treats**.

Similarly, Reinforcement Learning is:

- Training a computer to make a decision.
- By rewarding good choices and punishing bad ones.
- Just as you might train a dog with treats for learning tricks.

In reinforcement learning, there's an agent (for example a robot or computer program) that interacts with an environment. Let's take an example of teaching a computer program to play a game, for example chess.

- In this case, computer program is agent and chess game is the environment.
- The computer program can make different moves in the game, such as moving a chess piece.

- After each move, it receives feedback (reward or penalty) based on the outcome of the game.
- If it loses the game, it receives a negative reward, or a 'penalty'.
- Through trial and error, the program learns which moves lead to the best rewards, helping it figure out the best sequence of moves that leads to winning the game.

Reinforcement learning is powerful because it **allows machines to learn from their experiences** and make decisions in complex, uncertain environments — similar to how we learn from trial and error in the real world.

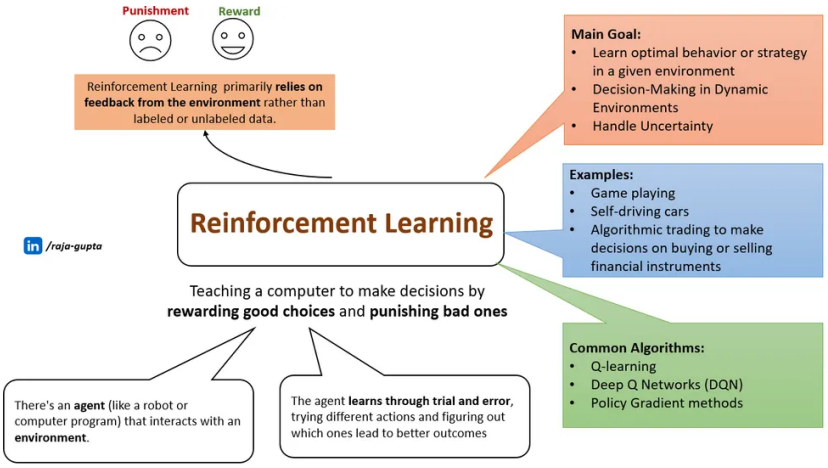


Figure 1.6: Reinforcement Learning summarized

# Chapter 2

# Algorithms

## 2.1 Regression Methods

Regression is one of the fundamental tasks in supervised learning. Its goal is to predict a continuous-valued output based on one or more input variables. Formally, given a dataset of input–output pairs $\{(x_i, y_i)\}_{i=1}^{N}$ where $x_i \in \mathbb{R}^d$ represents the input features and $y_i \in \mathbb{R}$ represents a continuous target variable, a regression model seeks to learn a function $f : \mathbb{R}^d \to \mathbb{R}$ that approximates the underlying relationship between inputs and outputs.

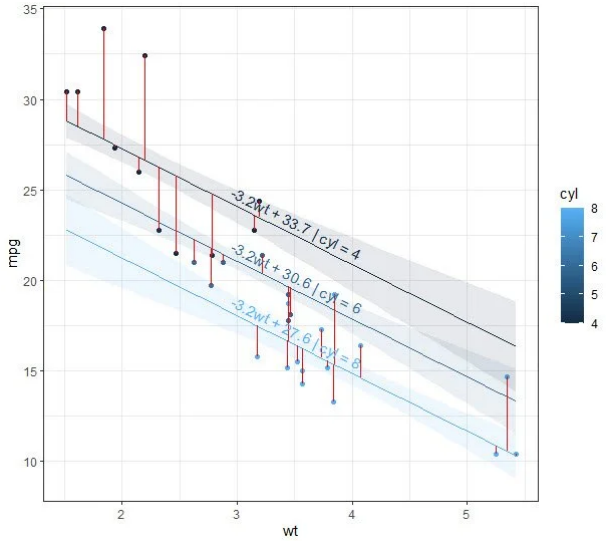


Figure 2.1: Regression examples

In a typical regression problem, the input can represent any structured or unstructured data — such as physical measurements, numerical attributes, or encoded representations of images, audio signals, or text. The model processes these inputs and outputs a single real number or, in the case of multivariate regression, a vector of continuous values.

The simplest and most widely known regression technique is *linear regression*, where the model assumes a linear relationship between the input features and the target:

$$\hat{y} = w^\top x + b,$$

where $w$ is a weight vector and $b$ is a bias term. More complex regression models, including polynomial regression, decision trees, and neural networks, can capture nonlinear dependencies between inputs and outputs.

The performance of a regression model is commonly evaluated using metrics such as Mean Squared Error (MSE), Root Mean Squared Error (RMSE), or Mean Absolute Error (MAE), which quantify how close the predicted values $\hat{y}_i$ are to the true targets $y_i$.

In summary, regression models map numerical representations of real-world phenomena to continuous outcomes, forming the basis for many predictive modeling tasks, from estimating housing prices and forecasting stock trends to predicting physical properties in scientific domains.

## 2.2 Classification Methods

Classification is another central task in supervised learning, where the objective is to assign each input to one of several discrete categories. Formally, given a dataset of labeled examples $\{(x_i, y_i)\}_{i=1}^{N}$ where $x_i \in \mathbb{R}^d$ represents the feature vector and $y_i \in \{1, \ldots, K\}$ denotes the class label, a classification model learns a function $f : \mathbb{R}^d \to \{1, \ldots, K\}$ or, equivalently, a probability distribution over classes $P(y|x)$.

The model ingests an input vector—which may represent text, an image, an audio signal, or any numerical encoding—and outputs either a class label (hard classification) or a vector of probabilities indicating the model's confidence for each class (soft classification). The most common approach for probabilistic outputs is to apply a softmax transformation to the model's final layer, ensuring that all predicted probabilities sum to one.

Several families of models can perform classification:

- **Linear Models:** Logistic Regression and Linear Discriminant Analysis (LDA) are among the simplest and most interpretable classifiers. They model class boundaries using linear decision surfaces and can provide calibrated probability estimates.
- **Support Vector Machines (SVM):** These models seek an optimal separating hyperplane that maximizes the margin between classes. Kernelized SVMs can capture nonlinear decision boundaries.
- **Tree-Based Models:** Decision Trees, Random Forests, and Gradient Boosting Machines (such as XGBoost or LightGBM) partition the feature space hierarchically. They often yield strong predictive performance and can handle heterogeneous data types.
- **Probabilistic Models:** Naïve Bayes and Gaussian Mixture Models (GMMs) represent classes using explicit probability distributions, allowing inference via Bayes' rule.
- **Neural Network Classifiers:** Deep architectures such as Multilayer Perceptrons (MLPs), Convolutional Neural Networks (CNNs), and Transformers are capable of learning highly nonlinear decision functions directly from raw data.

The quality of a classification model is typically assessed through metrics such as accuracy, precision, recall, F1-score, and the area under the Receiver Operating Characteristic curve (ROC–AUC). These measures quantify how effectively the model distinguishes among different classes and are essential for evaluating performance, particularly in the presence of class imbalance.

In summary, classification methods aim to map structured or unstructured input data to discrete output categories. From simple linear models to large-scale deep networks, the central challenge remains the same: to generalize from observed examples to unseen data, capturing the underlying structure that defines each class.

## 2.3 Clustering

Clustering is a fundamental technique in *unsupervised learning*. Its purpose is to discover inherent groupings within a dataset by identifying patterns or similarities among unlabeled examples. Unlike classification, where each data point is associated with a predefined label, clustering attempts to infer structure directly from the data itself. In this sense, it can be viewed as a way to explore the natural organization of data without prior supervision.

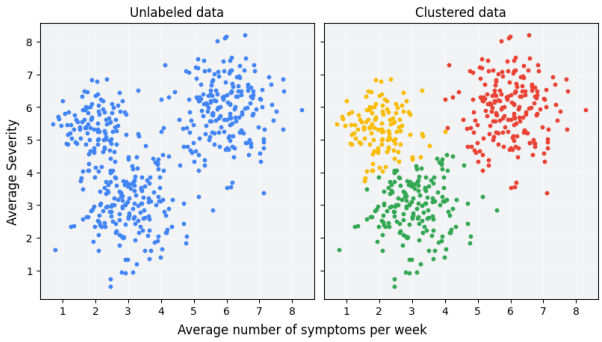


Figure 2.2: Unlabeled data vs Clustered data

Formally, given a set of examples $\{x_i\}_{i=1}^{N}$ with $x_i \in \mathbb{R}^d$, a clustering algorithm aims to partition the dataset into $K$ groups, or *clusters*, such that data points within the same cluster are more similar to each other than to those in different clusters. Similarity is typically measured through distance metrics such as Euclidean, Manhattan, or cosine distance, depending on the data representation and problem context.

To illustrate the concept, consider a clinical study designed to evaluate a new treatment protocol. During the experiment, patients report both the frequency and severity of their symptoms per week. Since these observations are continuous and not labeled, researchers can apply clustering analysis to group patients with similar responses to the treatment. For example, one cluster may contain patients who show rapid improvement, another may represent those with moderate progress, and a third may include patients with minimal or no response.

Figure **??** shows a simulated example in which patient data are grouped into three distinct clusters based on their symptom profiles. Such analyses allow researchers to uncover hidden patterns in clinical outcomes and to tailor future interventions according to the characteristics of each patient subgroup.

**Clustering Methods**

A variety of clustering algorithms have been developed, each suited to different data structures and assumptions:

- **K-Means Clustering:** A centroid-based algorithm that partitions data into $K$ clusters by minimizing the within-cluster sum of squared distances. It is simple and efficient, but assumes spherical cluster shapes and requires the number of clusters to be specified in advance.
- **Hierarchical Clustering:** Builds a nested hierarchy of clusters either by iteratively merging smaller clusters (agglomerative) or by splitting larger ones (divisive). The results are often visualized through a dendrogram, which provides insights into the data's structure at multiple levels of granularity.
- **Density-Based Clustering (DBSCAN, HDBSCAN):** Groups data points based on regions of high density, allowing the discovery of arbitrarily shaped clusters and the identification of noise or outliers. It does not require specifying the number of clusters a priori.
- **Gaussian Mixture Models (GMM):** A probabilistic approach in which each cluster is represented as a Gaussian distribution. Unlike K-Means, GMM provides *soft assignments*, meaning that each data point can belong to multiple clusters with different probabilities.
- **Spectral Clustering:** Utilizes the eigenvalues of a similarity matrix to perform clustering in a lower-dimensional space, making it suitable for non-convex or complex data manifolds.

The choice of clustering method depends on the data distribution, dimensionality, and the desired interpretability of results. In practice, clustering is often used as an exploratory tool for data analysis, feature engineering, or as a preprocessing step in larger machine learning pipelines.

In summary, clustering serves as a powerful approach to reveal latent structures within unlabeled datasets. It is widely applied in fields such as bioinformatics, market segmentation, image analysis, and recommender systems, where uncovering hidden patterns can lead to more meaningful understanding and informed decision-making.

## 2.4 Dimensionality Reduction

Dimensionality reduction is a key technique in machine learning and data analysis that seeks to represent high-dimensional data in a lower-dimensional space while preserving as much relevant information as possible. It serves multiple purposes: improving computational efficiency, mitigating the curse of dimensionality, reducing noise, and enhancing interpretability or visualization of complex datasets.

Formally, given a dataset $\{x_i\}_{i=1}^N$ where each example $x_i \in \mathbb{R}^D$ is described by $D$ features, the goal is to find a transformation function $f : \mathbb{R}^D \to \mathbb{R}^d$ with $d \ll D$ such that the lower-dimensional representation $z_i = f(x_i)$ retains the most informative characteristics

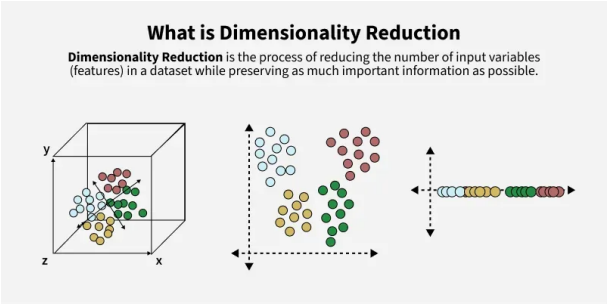


Figure 2.3: Dimensionality Reduction

of the original data. The challenge lies in compressing the representation without losing the essential structure or relationships between data points.

Dimensionality reduction techniques can be broadly divided into two categories: *feature selection* and *feature extraction*. Feature selection involves choosing a subset of the most relevant original variables, while feature extraction constructs new features as combinations or projections of the existing ones.

Several methods have been developed to perform dimensionality reduction, each based on different assumptions and optimization objectives:

- **Principal Component Analysis (PCA):** A linear technique that projects data onto the directions (principal components) of maximum variance. PCA identifies orthogonal axes that best explain the variability in the dataset and is often used for visualization and preprocessing.
- **Linear Discriminant Analysis (LDA):** Although originally designed for classification, LDA can also be used for dimensionality reduction by projecting data onto directions that maximize class separability.
- **Independent Component Analysis (ICA):** Aims to decompose multivariate data into statistically independent components, often used in signal processing and blind source separation tasks.
- **t-Distributed Stochastic Neighbor Embedding (t-SNE):** A nonlinear technique that preserves local similarities among data points, making it particularly useful for visualizing high-dimensional data in two or three dimensions.
- **Uniform Manifold Approximation and Projection (UMAP):** A more recent nonlinear method that approximates the topological structure of data manifolds, often providing better scalability and global structure preservation than t-SNE.
- **Autoencoders:** Neural network–based models that learn a compressed latent representation of data by training to reconstruct the input from a lower-dimensional code. They can capture complex nonlinear relationships and are widely used in deep learning pipelines.

To illustrate, consider a dataset describing patients with hundreds of medical features, such as laboratory values and genetic markers. Applying PCA or UMAP can reveal low-dimensional embeddings that group patients with similar clinical profiles, enabling researchers to visualize disease subtypes or predict treatment outcomes more effectively.

In summary, dimensionality reduction transforms complex, high-dimensional datasets into compact and informative representations. It facilitates efficient learning, improved generalization, and deeper insight into the intrinsic structure of data—serving as both a practical preprocessing step and a powerful tool for exploratory data analysis.

# Chapter 3

# Models

## 3.1 Neural Network Architectures

Neural network architectures define the structural organization and information flow within models that learn to approximate complex functions. They specify how neurons are arranged into layers, how these layers are connected, and how data propagate through the network during training and inference. Each architecture introduces specific inductive biases that make it particularly suited for certain types of data or tasks.

At their core, neural networks consist of a series of transformations applied to an input vector $x \in \mathbb{R}^d$. Each layer performs a linear mapping followed by a nonlinear activation function:

$$h^{(l)} = \sigma(W^{(l)}h^{(l-1)} + b^{(l)}),$$

where $W^{(l)}$ and $b^{(l)}$ are the layer's trainable parameters, $\sigma(\cdot)$ is a nonlinear activation function such as ReLU or GeLU, and $h^{(l)}$ represents the layer's output (often referred to as the hidden representation). The final layer produces an output $\hat{y}$ that corresponds to a prediction or learned embedding, depending on the task.

The design of a neural network architecture determines:

- **Depth:** the number of layers stacked sequentially, which influences the level of abstraction the model can learn.
- **Width:** the number of neurons per layer, affecting the model's capacity and expressiveness.
- **Connectivity:** how neurons or layers are connected—whether fully, locally, recurrently, or via attention mechanisms.
- **Parameter sharing and constraints:** architectural choices such as convolutions or recurrent weights that impose structure and reduce the number of learnable parameters.

The flexibility of neural networks lies in their ability to adapt architectural principles to specific problem domains. For instance, some architectures emphasize spatial feature extraction, others model temporal dependencies, and others learn global contextual relationships. This diversity allows neural networks to serve as a unifying framework across computer vision, natural language processing, speech recognition, and many other fields.

In practice, the choice of architecture depends on the nature of the input data and the desired output:

- For structured, tabular, or vector data, fully connected or feedforward networks are often employed.
- For grid-like data such as images, convolutional architectures are preferred.
- For sequential data such as time series or text, recurrent or attention-based architectures are more effective.

Although these categories differ in structure and function, they share common building blocks: linear transformations, nonlinear activations, normalization mechanisms, and regularization techniques. The interplay of these components defines the representational power and generalization ability of the network.

In summary, neural network architectures represent the blueprint of how information is processed and transformed within a model. By designing the right architecture, one can embed domain knowledge directly into the network's structure—guiding it to learn meaningful and efficient representations of data. The following sections present the most prominent architectural families—Feedforward, Convolutional, Recurrent, and Transformer networks—each tailored to specific data modalities and learning paradigms.

## 3.2 Feedforward Networks such as Multi Layer Perceptron (MLP)

Feedforward Neural Networks, commonly referred to as *Multilayer Perceptrons (MLPs)*, represent the simplest and most fundamental class of neural network architectures. They serve as the conceptual foundation upon which more advanced architectures—such as Convolutional, Recurrent, and Transformer models—are built. An MLP processes information strictly in one direction: from input to output, without any recurrent or feedback connections.

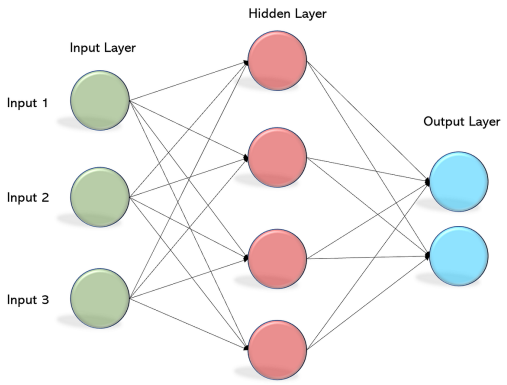


Figure 3.1: MLP

### 3.2.1 Structure and Forward Propagation

An MLP is composed of three main components:

- **Input layer:** receives the raw data, where each neuron corresponds to one input feature.
- **Hidden layers:** perform a sequence of linear and nonlinear transformations to extract intermediate representations.
- **Output layer:** produces the final prediction, whose dimensionality depends on the task (e.g., a single neuron for regression, or multiple neurons for classification).

Each layer in the network is fully connected to the next, meaning that every neuron in one layer influences every neuron in the subsequent layer through weighted connections. Information flows forward through the network, with no cycles or memory of previous inputs.

The data propagate through the network according to the following computation:

$$h^{(1)} = \sigma(W^{(1)}x + b^{(1)}), \quad h^{(2)} = \sigma(W^{(2)}h^{(1)} + b^{(2)}), \quad \hat{y} = f(W^{(3)}h^{(2)} + b^{(3)}),$$

where $x$ represents the input vector, $W^{(l)}$ and $b^{(l)}$ are the trainable weights and biases of layer $l$, $\sigma(\cdot)$ is a nonlinear activation function (such as ReLU or GeLU), and $\hat{y}$ is the final network output.

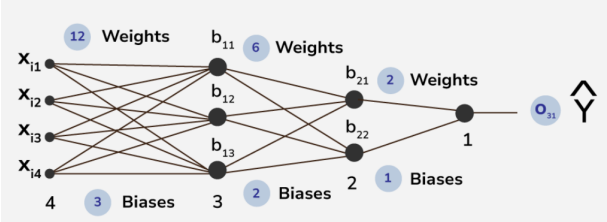


Figure 3.2: Forward Propagation

The use of nonlinear activation functions between layers allows MLPs to model complex, non-linear relationships between inputs and outputs. Without these nonlinearities, the composition of multiple layers would collapse into a single linear transformation, greatly limiting the model's expressive power.

### 3.2.2 Mathematical Formulation

Formally, a feedforward network defines a parameterized function $f_\theta(x)$ that maps an input $x \in \mathbb{R}^d$ to an output $\hat{y} \in \mathbb{R}^k$:

$$f_\theta(x) = W^{(L)}\,\sigma\big(W^{(L-1)}\,\sigma\big(\cdots\sigma(W^{(1)}x + b^{(1)})\big) + b^{(L-1)}\big) + b^{(L)}.$$

Here, $\theta = \{W^{(l)}, b^{(l)}\}_{l=1}^{L}$ represents all learnable parameters of the network. The number of layers $L$ determines the network's depth, while the number of neurons per layer determines its width and representational capacity.

The *Universal Approximation Theorem* states that a feedforward network with at least one hidden layer and a non-linear activation function can approximate any continuous function on a compact subset of $\mathbb{R}^n$, given sufficient neurons. This theoretical result underscores the expressive power of even simple MLPs.

### 3.2.3 Training Process

Training an MLP involves adjusting its parameters $\theta$ to minimize a predefined loss function that quantifies the discrepancy between predicted outputs $\hat{y}$ and true targets $y$. This optimization is typically performed using stochastic gradient descent (SGD) or one of its adaptive variants such as Adam or RMSProp. Gradients of the loss function with respect to each parameter are computed efficiently using the *backpropagation* algorithm.

During training, techniques such as *Dropout*, *Batch Normalization*, and *Weight Regularization* are commonly applied to improve generalization and prevent overfitting. The learning rate, batch size, and number of hidden layers are hyperparameters that must be tuned based on the complexity of the dataset and the desired performance.

### 3.2.4 Applications and Limitations

Feedforward networks are versatile and can be applied to a wide range of problems, including:

- **Regression tasks:** predicting continuous outcomes such as prices, temperatures, or sensor readings.
- **Classification tasks:** assigning input samples to discrete categories.
- **Representation learning:** serving as encoders or decoders in larger architectures such as autoencoders.

However, MLPs treat all input features as independent and lack mechanisms to exploit structural information such as spatial or temporal relationships. This limitation motivates the development of specialized architectures like Convolutional Neural Networks (CNNs) for spatial data and Recurrent Neural Networks (RNNs) for sequential data.

In summary, Feedforward Networks and MLPs form the conceptual foundation of modern deep learning. They introduce the principles of layered computation, parameter optimization, and nonlinear transformation that underpin all subsequent neural architectures. Although simple in structure, they remain powerful tools for modeling generic data and continue to play a central role in hybrid and advanced neural systems.

## 3.3 Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNNs) represent a specialized class of neural architectures designed to process data with a grid-like topology, such as images, audio spectrograms, or video frames. They extend the concept of feedforward networks by introducing spatially local connections and shared weights, allowing them to efficiently capture hierarchical and translationally invariant features within structured inputs.

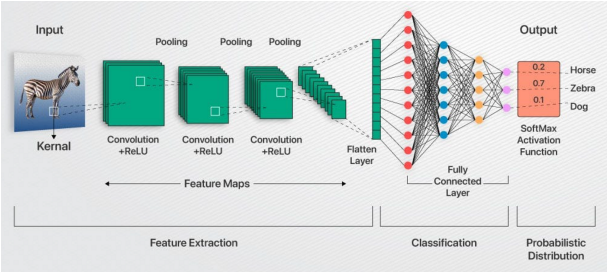


Figure 3.3: CNN

### 3.3.1 Motivation and Overview

Traditional feedforward networks treat all input features as independent and require a large number of parameters to process high-dimensional data, such as pixels in an image. CNNs address this limitation by exploiting the spatial correlation between neighboring input elements. Instead of connecting each neuron to every input unit, convolutional layers connect each neuron only to a small region of the input—called the *receptive field*.

This localized connectivity enables CNNs to detect low-level features (e.g., edges, corners, or textures) in early layers, and progressively more complex patterns (e.g., shapes or objects) in deeper layers. As a result, CNNs can learn compact and meaningful feature hierarchies with far fewer parameters than fully connected networks.

### 3.3.2 Convolutional Operations

The core operation of a CNN is the *convolution*, which applies a set of learnable filters (kernels) to the input data. For a 2D input image $I$ and a filter $K$ of size $m \times n$, the convolution operation is defined as:

$$S(i,j) = (I * K)(i,j) = \sum_{u=0}^{m-1} \sum_{v=0}^{n-1} I(i+u, j+v)\, K(u,v),$$

where $S(i,j)$ denotes the output feature map (also known as an activation map). Each filter extracts a specific type of feature, and multiple filters operating in parallel produce several feature maps that capture different aspects of the input.

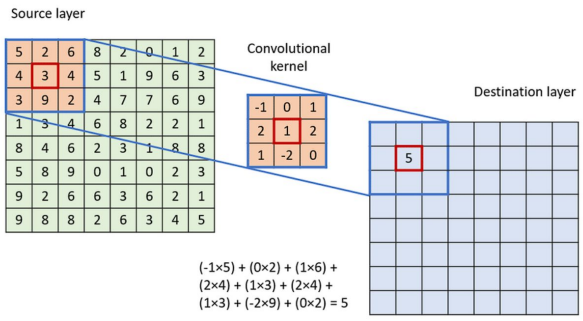


Figure 3.4: Convolution Illustration

After each convolutional layer, nonlinear activation functions (such as ReLU) introduce nonlinearity, enabling the network to model complex relationships between features.

These layers are often followed by pooling operations that downsample the spatial dimensions, reducing computational cost and increasing translational robustness.

### 3.3.3 Architectural Components

A typical CNN architecture consists of several types of layers arranged in sequence:

- **Convolutional layers:** perform feature extraction using multiple filters. The output of each convolution represents the response of a particular feature detector applied across the spatial domain.
- **Pooling layers:** reduce the spatial dimensions of feature maps by summarizing local regions (e.g., using max or average pooling), improving computational efficiency and spatial invariance.
- **Normalization layers:** such as Batch Normalization, stabilize and accelerate training by standardizing intermediate activations.
- **Fully connected layers:** often used at the end of the network to combine the extracted features into a final decision (e.g., class probabilities).

Early CNN architectures such as *LeNet-5* demonstrated the viability of this approach for digit recognition, while later models such as *AlexNet*, *VGGNet*, and *GoogLeNet* introduced deeper hierarchies and more efficient filter designs. Modern CNNs like *ResNet* employ *residual connections*—direct skip links between layers—to facilitate gradient flow and enable the training of extremely deep networks without performance degradation.

### 3.3.4 Training and Optimization

Training a CNN follows the same principles as other neural networks: parameters are optimized to minimize a loss function (e.g., cross-entropy for classification) using stochastic gradient descent (SGD) or adaptive optimizers such as Adam. However, CNN training introduces specific challenges related to overfitting, computational cost, and vanishing gradients.

To mitigate these issues, practitioners commonly employ:

- **Data augmentation:** artificially increasing the diversity of training data through transformations such as rotation, scaling, or flipping.
- **Regularization techniques:** such as Dropout and Weight Decay to improve generalization.
- **Batch Normalization:** to stabilize learning dynamics and accelerate convergence.
- **Transfer learning:** reusing pretrained CNNs (e.g., ResNet, VGG) as feature extractors for new tasks, significantly reducing training time and required data.

### 3.3.5 Applications and Impact

CNNs have become the cornerstone of modern computer vision and beyond. Their ability to learn spatial hierarchies of features has led to breakthroughs in tasks such as:

- Image classification and object detection (e.g., ImageNet, YOLO, Faster R-CNN)
- Semantic and instance segmentation (e.g., U-Net, Mask R-CNN)
- Image generation and restoration (e.g., autoencoders, GANs)
- Medical imaging, remote sensing, and visual inspection

Beyond vision, CNNs have also been successfully adapted to non-visual domains such as audio analysis, natural language processing, and time-series forecasting, where local dependencies and translation-invariant features are relevant.

In summary, Convolutional Neural Networks introduced the concept of local receptive fields and parameter sharing, revolutionizing how deep models handle structured data. Their hierarchical feature extraction and scalability make them one of the most influential architectures in the history of machine learning, forming the basis for numerous modern deep learning systems.

## 3.4 Recurrent and Sequential Models (RNN, LSTM, GRU)

## 3.5 Transformer Architectures and Attention Mechanisms

## 3.6 Core Components and Techniques

## 3.7 Activation Functions (ReLU, GeLU, Sigmoid, Tanh)

## 3.8 Linear and Dense Layers (e.g., nn.Linear)

## 3.9 Normalization Layers (BatchNorm, LayerNorm, GroupNorm)

## 3.10 Dropout and Regularization Techniques

## 3.11 Residual Connections and Skip Layers

## 3.12 Weight Initialization and Parameterization

## 3.13 Training and Optimization

## 3.14 Loss Functions (Cross-Entropy, MSE, etc.)

## 3.15 Optimization Algorithms (SGD, Adam, RMSProp)

Optimization algorithms govern how neural network parameters are updated to minimize a loss function. They determine the direction and magnitude of each update and thus directly affect convergence speed, stability, and final generalization. Below, we present a concise yet rigorous overview of Stochastic Gradient Descent (SGD), RMSProp, and Adam—explaining their mechanics, intuition, and when each is preferable.

### 3.15.1 From Gradient Descent to Stochastic Updates

Given parameters $\theta$ and loss $\mathcal{L}(\theta)$, *(full-batch)* gradient descent updates:

$$\theta_{t+1} \;=\; \theta_t \;-\; \eta\, \nabla_\theta \mathcal{L}(\theta_t),$$

where $\eta$ is the learning rate. In deep learning, we approximate the gradient using mini-batches $B_t$:

$$g_t \;=\; \nabla_\theta \mathcal{L}_{B_t}(\theta_t), \qquad \theta_{t+1} \;=\; \theta_t - \eta\, g_t,$$

which yields *Stochastic Gradient Descent (SGD)*—noisier but vastly more efficient on large datasets.

### 3.15.2 SGD (with Momentum and Nesterov)

**Vanilla SGD.** SGD performs small steps along the negative gradient. It is simple, memory-light, and often yields strong generalization, but raw SGD can be slow in narrow valleys and sensitive to learning-rate choice.

**SGD with Momentum.** To accelerate along consistent directions and damp oscillations, momentum accumulates an *exponential moving average* of past gradients:

$$v_t \;=\; \mu\, v_{t-1} + (1-\mu)\, g_t, \qquad \theta_{t+1} \;=\; \theta_t - \eta\, v_t,$$

where $v_t$ is the velocity and $\mu \in [0, 1)$ is the momentum coefficient (e.g., 0.9). Intuition: like pushing a heavy ball down a slope—small, consistent pushes add up to faster progress.

**Nesterov Momentum (NAG).** Looks ahead before computing the gradient, often yielding smoother convergence:

$$\tilde{\theta}_t \;=\; \theta_t - \eta\,\mu\, v_{t-1}, \qquad v_t \;=\; \mu\, v_{t-1} + (1-\mu)\,\nabla_\theta \mathcal{L}_{B_t}(\tilde{\theta}_t), \qquad \theta_{t+1} \;=\; \theta_t - \eta\, v_t.$$

**When to implement SGD.**

- **Strong baseline & generalization:** In vision tasks (e.g., CNNs), SGD(+momentum) often reaches better generalization than adaptive methods.
- **Stable loss landscapes:** When gradients are relatively well-scaled (after normalization), SGD can be very effective.
- **Low memory footprint:** Minimal optimizer state.

**A simple example.** Minimize $f(\theta) = \frac{1}{2} a\theta^2$ with $a > 0$. SGD update: $\theta_{t+1} = \theta_t - \eta a \theta_t = (1 - \eta a)\theta_t$. If $\eta$ is too large, $(1 - \eta a)$ oscillates or diverges; with momentum, updates dampen oscillations along steep directions (large $a$).

### 3.15.3 RMSProp

**Idea.** RMSProp adapts the learning rate per-parameter by normalizing the gradient with a running average of its squared values—mitigating issues where some parameters consistently have larger gradients.

**Update rule.** With decay $\beta \in (0, 1)$ (e.g., 0.9) and small $\epsilon$ (e.g., $10^{-8}$):

$$s_t \;=\; \beta\, s_{t-1} + (1-\beta)\, g_t^2, \qquad \theta_{t+1} \;=\; \theta_t \;-\; \frac{\eta}{\sqrt{s_t} + \epsilon}\, g_t.$$

Here $s_t$ tracks recent gradient magnitudes; parameters with consistently large gradients get effectively smaller step sizes, and vice versa.

**When to implement RMSProp.**

- **Non-stationary, noisy problems:** Historically popular for RNNs and reinforcement learning where gradient scales change over time.
- **Unevenly scaled features:** Works well when different parameters exhibit very different gradient magnitudes.
- **Fast tuning:** Fewer LR sweeps needed compared to raw SGD.

**Toy intuition.** If one parameter sees gradients about $10\times$ larger than another, RMSProp will downscale its effective step, helping keep updates balanced without manual per-parameter tuning.

### 3.15.4 Adam (and AdamW)

**Idea.** Adam combines momentum (first-moment averaging) with RMSProp-like scaling (second-moment averaging), plus bias corrections—yielding robust, per-parameter adaptive steps.

**Update rule.** With $(\beta_1, \beta_2)$ (e.g., $0.9, 0.999$) and $\epsilon$:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2,$$

Bias-corrected estimates:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t},$$

Parameter update:

$$\theta_{t+1} \;=\; \theta_t \;-\; \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\, \hat{m}_t.$$

**AdamW (decoupled weight decay).** Standard Adam mixes $L_2$ regularization into the adaptive scaling, which can behave differently from true weight decay. AdamW decouples the decay:

$$\theta_{t+1} \;=\; (1 - \eta\lambda)\, \theta_t \;-\; \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\, \hat{m}_t,$$

where $\lambda$ is the weight decay coefficient—often preferred in modern deep nets (e.g., Transformers).

**When to implement Adam/AdamW.**

- **Strong default for many tasks:** Fast convergence and ease of tuning.
- **Sparse or heavy-tailed gradients:** NLP, Transformers, embeddings—adaptive steps help a lot.
- **Heterogeneous scales:** Different layers/parameters with widely varying gradient magnitudes.

**Caveats.**

- **Generalization gap:** On some vision tasks, Adam may converge faster but generalize slightly worse than tuned SGD(+momentum).
- **Tune $\epsilon$ if needed:** Can stabilize training when variances are tiny.
- **Prefer AdamW:** Decoupled weight decay tends to yield more predictable regularization.

### 3.15.5 Choosing Between SGD, RMSProp, and Adam

- **Use SGD (+Momentum/Nesterov)** when you care about *final generalization* and have reasonably normalized inputs (common in CNN-based vision). It is memory-efficient and often wins with enough tuning (LR, momentum, schedule).
- **Use RMSProp** for non-stationary or very noisy objectives (e.g., some RL or RNN settings) and when gradient scales drift significantly over training.
- **Use Adam/AdamW** as a strong, practical default—particularly in NLP/Transformers, multimodal models, and problems with sparse or uneven gradients—fast to good performance with modest tuning.

### 3.15.6 Hyperparameter Heuristics and Practical Tips

- **Defaults:** Adam/AdamW: $\eta$=$10^{-3}$, $(\beta_1, \beta_2)$=$(0.9, 0.999)$, $\epsilon$=$10^{-8}$; RMSProp: $\eta$=$10^{-3}$, $\beta$=0.9, $\epsilon$=$10^{-8}$; SGD: start $\eta \in [10^{-2}, 10^{-1}]$, momentum 0.9.
- **Schedule the LR:** Step decay or cosine annealing often improves all three optimizers.
- **Normalize/standardize inputs:** Helps SGD; still beneficial for adaptive methods.
- **Batch size coupling:** Larger batches often need larger learning rates (linear scaling rule).
- **Regularization:** Prefer decoupled weight decay (AdamW) over $L_2$ in Adam; with SGD use weight decay and/or dropout.

### 3.15.7 Intuition via a Simple 2D Bowl

Consider minimizing $f(\theta_1, \theta_2) = \frac{1}{2}(a\,\theta_1^2 + b\,\theta_2^2)$ with $a \gg b$ (a narrow, steep valley).

- **SGD** zig-zags across the steep axis unless momentum is used; with momentum it accelerates along the shallow axis and damps oscillations along the steep one.

- **RMSProp** scales steps by recent squared gradients, shrinking steps along the steep axis (large gradients) and enlarging along the shallow axis—quickly balancing progress.
- **Adam** couples momentum (directional memory) with RMSProp-like scaling, often moving decisively toward the minimum with stable steps; AdamW improves regularization on top.

### 3.15.8 Summary

SGD, RMSProp, and Adam represent a progression from simple, global step sizes to adaptive, per-parameter updates with momentum. **SGD(+momentum)** remains a gold standard for strong generalization (especially in vision). **RMSProp** addresses non-stationary, uneven gradient scales. **Adam/AdamW** is a robust default for many modern architectures, particularly where gradients are sparse or heterogeneous. Selecting among them—and combining with a sound learning-rate schedule—often determines whether training is merely adequate or state-of-the-art.

## 3.16 Learning Rate Scheduling

The *learning rate* is one of the most fundamental hyperparameters in the optimization of neural networks. It determines the size of the steps taken by the optimizer when updating the model's parameters in response to the computed gradients. In simple terms, the learning rate controls how quickly or cautiously a model learns from the training data.

At each iteration of gradient-based optimization, the model parameters $\theta$ are updated according to:

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta_t} \mathcal{L}(\theta_t),$$

where $\mathcal{L}(\theta_t)$ is the loss function, $\nabla_{\theta_t} \mathcal{L}$ is the gradient of the loss with respect to the parameters, and $\eta$ is the learning rate. A larger $\eta$ means taking bigger steps in the direction that reduces the loss, while a smaller $\eta$ means taking smaller, more cautious steps.

The learning rate directly affects the trajectory of the optimization process through the loss landscape:

- If the learning rate is **too high**, the optimization steps overshoot the minima, leading to oscillations or even divergence of the loss.
- If it is **too low**, progress becomes extremely slow, and the model might get trapped in shallow minima or plateaus.

Thus, selecting an appropriate learning rate is a critical factor in achieving efficient and stable convergence. However, a single static value rarely performs optimally throughout training—different phases of learning benefit from different update magnitudes. This observation motivates the use of *learning rate scheduling*.

### 3.16.1 Theoretical Overview of Learning Rate Scheduling

*Learning rate scheduling* refers to the process of dynamically adjusting the learning rate $\eta_t$ as training progresses. Early in training, a relatively large learning rate helps the model explore the loss surface and escape poor local minima. Later in training, a smaller learning rate allows for more precise fine-tuning as the optimizer approaches convergence.

Mathematically, the update rule with a time-dependent learning rate becomes:

$$\theta_{t+1} = \theta_t - \eta_t \nabla_{\theta_t} \mathcal{L}(\theta_t),$$

where $\eta_t$ evolves according to a scheduling policy. Different scheduling strategies define how $\eta_t$ changes over epochs or iterations.

Common scheduling methods include:

- **Step Decay:** Reduces the learning rate by a constant factor after a fixed number of epochs:
$$\eta_t = \eta_0 \cdot \gamma^{\lfloor t/T \rfloor}.$$
- **Exponential Decay:** Decreases the learning rate continuously as $\eta_t = \eta_0 \cdot e^{-\lambda t}$, providing a smooth decay.
- **Cosine Annealing:** Gradually lowers the learning rate following a cosine curve:
$$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min}) \left(1 + \cos\left(\frac{t}{T_{\max}}\pi\right)\right).$$
- **Cyclical Learning Rate (CLR):** Periodically increases and decreases $\eta_t$ between lower and upper bounds to escape local minima.
- **One-Cycle Policy:** Increases the learning rate rapidly at the start of training, then gradually reduces it below the initial value, promoting fast convergence and strong generalization.

A *warm-up* phase is often included, during which the learning rate starts small and increases linearly or exponentially for the first few epochs, stabilizing early training—especially in deep architectures such as Transformers.

### 3.16.2 Influence of the Learning Rate on Model Training

The learning rate profoundly influences how a neural network learns. Its value and evolution determine whether training will be stable, efficient, or divergent.

- **High Learning Rate:** When $\eta$ is too large, parameter updates overshoot the optimal region, causing oscillations or divergence. The model may fail to minimize the loss and instead fluctuate around high-error regions. While a large learning rate accelerates initial progress, it often prevents fine convergence.
- **Low Learning Rate:** If $\eta$ is too small, updates become negligible, and training proceeds very slowly. The model may converge prematurely to suboptimal minima or fail to adapt effectively, leading to underfitting.

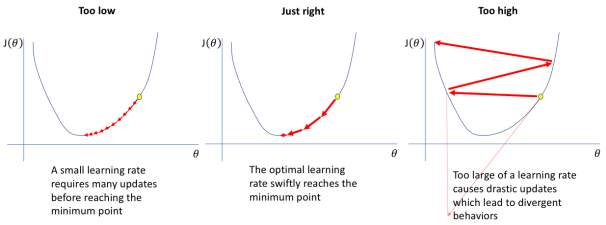


Figure 3.5: Learning Rate explained

- **Dynamic Scheduling:** A learning rate schedule allows a controlled transition—from fast exploration to fine-grained optimization. Larger learning rates early on encourage broad exploration of the loss surface, while progressively smaller rates near convergence allow the model to refine weights precisely. This results in both faster training and improved generalization.

The relationship between learning rate and other hyperparameters—such as batch size, momentum, and weight decay—is also crucial. For instance, larger batch sizes often require proportionally higher learning rates to maintain gradient stability, a phenomenon described by the *linear scaling rule*.

### 3.16.3 Practical Considerations

In practice, identifying an optimal learning rate or schedule typically involves empirical experimentation. A common approach is the *learning rate finder*, which gradually increases the learning rate during a brief training run and records the loss. Plotting loss against learning rate reveals a region where the loss decreases most rapidly—indicating an effective initial value.

Deep learning frameworks such as PyTorch and TensorFlow provide built-in tools for scheduling, including: `StepLR`, `ExponentialLR`, `ReduceLROnPlateau`, and `CosineAnnealingLR`. These utilities automate the adjustment of $\eta_t$ during training, making learning rate scheduling a standard part of modern optimization pipelines.

In summary, the learning rate defines the pace of learning in neural network training—too high leads to instability, too low to stagnation. Learning rate scheduling refines this process by allowing the model to adaptively balance exploration and convergence, resulting in faster, more stable, and more accurate learning.

# Chapter 4

# Metrics & how to read it

## 4.1 Accuracy & Loss

During the training of a machine learning model, *accuracy* and *loss* are two key metrics used to assess how well the model is learning and generalizing. Together, they provide complementary insights: accuracy measures the proportion of correct predictions, while loss quantifies the numerical difference between the predicted and true outputs. Monitoring these metrics over time allows practitioners to diagnose issues such as underfitting, overfitting, or poor optimization behavior.

Accuracy and loss are typically visualized as curves across training epochs. The shape and trends of these curves reveal how the model's performance evolves during training and validation, helping to identify whether the model is improving, stagnating, or degrading.

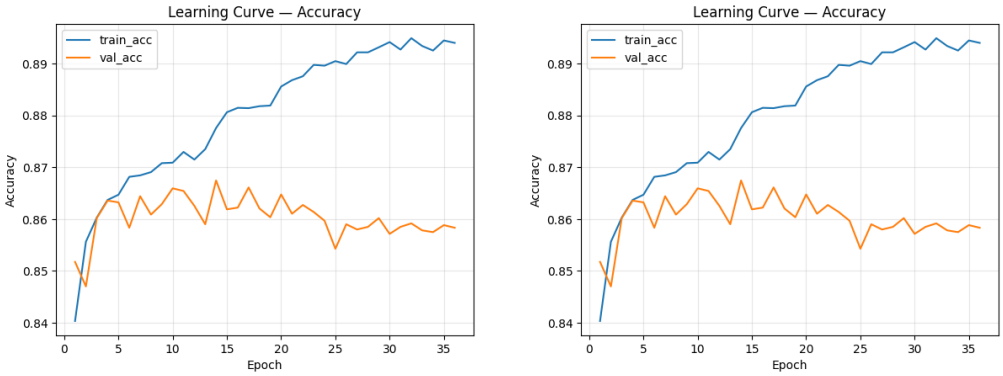


Figure 4.1: Learning Curve for Loss and Accuracy

### 4.1.1 Accuracy

Accuracy measures the fraction of correctly predicted samples relative to the total number of samples. For a classification task, it is formally defined as:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}.$$

A high accuracy indicates that the model correctly classifies most examples in the dataset. However, accuracy alone can be misleading—especially in imbalanced datasets where some

classes are more frequent than others. In such cases, complementary metrics such as precision, recall, and F1-score provide a more balanced evaluation.

The *accuracy curve* plots model accuracy over successive training epochs for both training and validation sets. A steadily increasing training accuracy accompanied by stable or improving validation accuracy indicates effective learning. Conversely, if validation accuracy stagnates or declines while training accuracy continues to rise, the model may be overfitting.

### 4.1.2 Loss

Loss quantifies how far the model's predictions are from the true target values. It is computed using a *loss function* (or *cost function*) that assigns a penalty to incorrect or inaccurate predictions. The goal of training is to minimize this loss function through iterative optimization, typically via gradient descent or one of its variants.

For example, in regression tasks, the most common loss function is the *Mean Squared Error (MSE)*, defined as:

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

where $y_i$ represents the true value and $\hat{y}_i$ the predicted output. In classification tasks, loss functions such as *Cross-Entropy Loss* or *Negative Log-Likelihood* are widely used to measure the divergence between the predicted probability distribution and the true labels.

The *loss curve* typically decreases as training progresses, indicating that the model is minimizing its prediction errors. However, a widening gap between training and validation loss often signals overfitting, whereas consistently high loss on both sets may indicate underfitting or inappropriate model complexity.

In summary, accuracy and loss are fundamental diagnostic tools in machine learning. Their joint analysis provides valuable feedback on model convergence, learning efficiency, and generalization performance, guiding practitioners in fine-tuning architectures and optimization strategies.

## 4.2 Precision & Recall

While accuracy provides a general indication of model performance, it can be misleading in situations where data are imbalanced—when certain classes occur much more frequently than others. In such cases, metrics like *precision* and *recall* offer a more informative and nuanced evaluation of classification models, especially in binary or multi-class settings where the cost of false positives and false negatives differs.

Precision and recall are often analyzed together to assess how well a model identifies positive instances without introducing excessive misclassifications. Their trade-off is frequently visualized through a *precision–recall curve*, which illustrates the relationship between these two quantities as the decision threshold varies.

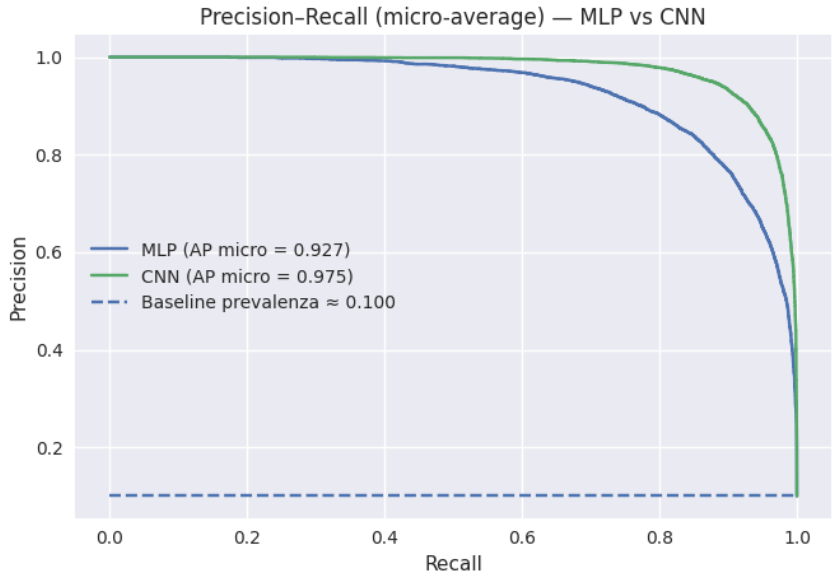


Figure 4.2: Precision Recall Curve

### 4.2.1 Precision

Precision measures the proportion of positive predictions that are actually correct. It answers the question: *Of all instances predicted as positive, how many are truly positive?* Mathematically, precision is defined as:

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}}.$$

High precision indicates that the model makes few false positive errors—it is "cautious" in labeling samples as positive. However, a model with very high precision may miss many actual positives if it becomes too conservative, leading to low recall.

In the precision–recall curve, precision typically decreases as recall increases. This occurs because, as the model becomes more inclusive (labeling more instances as positive), it also tends to introduce more false positives.

### 4.2.2 Recall

Recall, also known as *sensitivity* or *true positive rate*, measures the proportion of actual positives that the model successfully identifies. It answers the question: *Of all true positive instances, how many did the model correctly detect?* Formally, recall is defined as:

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}.$$

A high recall value indicates that the model captures most of the true positives but may include more false positives. In many real-world applications—such as medical diagnosis or fraud detection—high recall is critical, as missing a positive case can be costly or dangerous.

### 4.2.3 Confusion Matrix

The *confusion matrix* is a fundamental tool for evaluating the performance of classification models. It provides a detailed breakdown of the model's predictions compared to the actual ground truth labels, allowing practitioners to understand not only how many predictions were correct, but also the types of errors the model made.

In a binary classification setting, the confusion matrix is a $2 \times 2$ table that summarizes the counts of true and false predictions for both classes:

| | **Predicted Positive** | **Predicted Negative** |
|---|---|---|
| **Actual Positive** | True Positive (TP) | False Negative (FN) |
| **Actual Negative** | False Positive (FP) | True Negative (TN) |

Here:

- **True Positives (TP):** Instances correctly classified as belonging to the positive class.
- **True Negatives (TN):** Instances correctly classified as belonging to the negative class.
- **False Positives (FP):** Instances incorrectly classified as positive (Type I error).
- **False Negatives (FN):** Instances incorrectly classified as negative (Type II error).

From the confusion matrix, a variety of performance metrics can be derived, including:

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

These relationships make the confusion matrix an essential foundation for most evaluation metrics in classification problems.

For *multi-class classification*, the confusion matrix extends to an $N \times N$ table, where $N$ is the number of classes. Each row represents the actual class, and each column represents the predicted class. Diagonal entries correspond to correct predictions, while off-diagonal entries indicate misclassifications between specific classes. Visualizing the confusion matrix as a heatmap can help reveal systematic biases—for instance, when the model consistently confuses certain categories due to similarity in features or insufficient training examples.

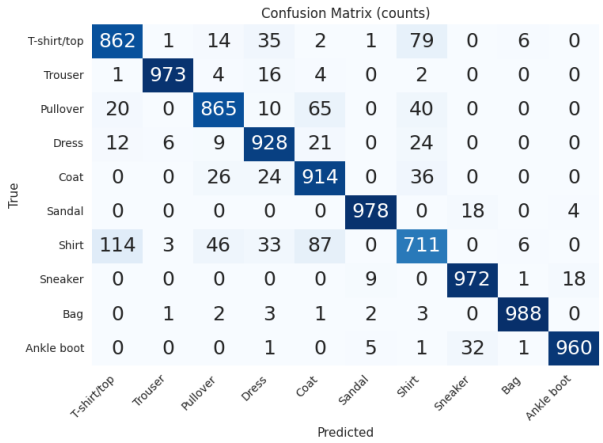


Figure 4.3: Confusion Matrix for fashion MNIST dataset

In practice, confusion matrices are often used alongside precision–recall and ROC curves to provide a comprehensive understanding of a model's behavior. They allow practitioners to identify which types of errors are most common, assess the balance of performance across classes, and guide future steps in data collection, model design, or hyperparameter tuning.

In summary, the confusion matrix is not merely a diagnostic tool but a window into the decision-making process of a classifier. It enables a granular analysis of model performance, transforming raw prediction counts into actionable insights for model refinement and validation.

### 4.2.4 Precision–Recall Trade-off and Curve Interpretation

Precision and recall are inherently linked: improving one often comes at the expense of the other. By adjusting the model's decision threshold, one can balance the two metrics depending on the specific requirements of the application. The *precision–recall curve* plots recall on the x-axis and precision on the y-axis, summarizing performance across all possible thresholds.

A model that perfectly separates classes would achieve a precision of 1.0 across all recall levels, resulting in a flat line at the top of the plot. In contrast, a random classifier produces a curve close to the baseline, indicating that precision falls rapidly as recall increases. The area under the precision–recall curve (AUC–PR) provides a scalar summary of overall performance, especially useful when comparing different models on imbalanced datasets.

In summary, precision and recall offer complementary perspectives on model performance. Precision emphasizes correctness among positive predictions, while recall emphasizes completeness in capturing all positive instances. Together, they form the foundation for the *F1-score*, a harmonic mean that provides a single, balanced measure of both precision and recall—allowing for a more holistic evaluation of classification quality.

## 4.3 F1-Score

The *F1-score* is a widely used metric that combines *precision* and *recall* into a single measure of a model's effectiveness. It is particularly valuable when dealing with imbalanced datasets, where accuracy alone can be misleading, and when both false positives and false negatives carry significant consequences.

Precision measures how many of the predicted positive instances are truly positive, while recall measures how many of the actual positive instances were correctly identified. Since these two metrics often trade off against each other, the F1-score provides a balanced summary by computing their harmonic mean:

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}.$$

This formulation ensures that a high F1-score can only be achieved when both precision and recall are reasonably high. If either precision or recall is low, the F1-score will also decrease sharply, making it more sensitive to performance imbalances than a simple arithmetic mean.

The F1-score ranges between 0 and 1, where 1 indicates perfect precision and recall, and 0 indicates the worst possible performance. It is especially useful when the cost of false positives and false negatives is roughly equal, or when the dataset exhibits skewed class distributions.

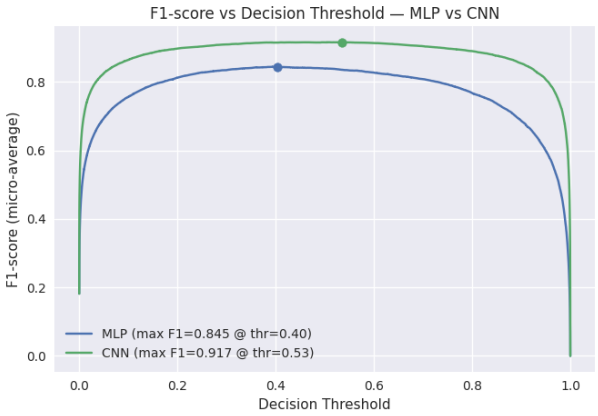


Figure 4.4: F1 score graph with threshold

In a binary classification context, a single F1-score value summarizes model performance. However, for multi-class problems, it can be generalized using different averaging strategies:

- **Macro F1-score:** Computes the F1-score independently for each class and then takes the unweighted mean. It treats all classes equally, regardless of their frequency.
- **Micro F1-score:** Aggregates all true positives, false positives, and false negatives across classes before computing the F1-score. It gives more weight to frequent classes.
- **Weighted F1-score:** Computes the class-specific F1-scores and averages them proportionally to the number of true instances in each class. This approach balances performance evaluation according to class importance.

The *F1-score curve* can also be plotted against varying classification thresholds to visualize the trade-off between precision and recall. By adjusting the decision threshold, practitioners can tune the balance between sensitivity (recall) and specificity (precision) to best fit the requirements of a specific application—such as maximizing recall in medical diagnosis or maximizing precision in spam detection.

In summary, the F1-score provides a robust and interpretable single-number metric for evaluating classification performance, especially in the presence of imbalanced data. It captures the delicate equilibrium between precision and recall, offering a more truthful representation of model effectiveness than accuracy alone. For this reason, it is a standard metric in research benchmarks and real-world machine learning evaluations alike.

# Chapter 5

# Implementation & Code Patterns

# Chapter 6

# Advanced Topics