Artificial Neural Networks and Deep Learning - Notes - v0.2.0

260236

October 2025

Preface

Every theory section in these notes has been taken from two sources:

- Ian Goodfellow and Yoshua Bengio and Aaron Courville, Deep Learning, MIT Press. [1]
- Course slides. [3]

About:

GitHub repository

These notes are an unofficial resource and shouldn't replace the course material or any other book on artificial neural networks and deep learning. It is not made for commercial purposes. I've made the following notes to help me improve my knowledge and maybe it can be helpful for everyone.

As I have highlighted, a student should choose the teacher's material or a book on the topic. These notes can only be a helpful material.

Contents

1 Introduction to Deep Learning

1.1 Machine Learning Foundations

Humans and animals learn from experience. Computers, too, can improve performance when exposed to more data or feedback. But how do we formally define "learning" in a way that's precise enough for a engineering course? Tom Mitchell[1], in 1997, proposed a now-classic definition:

Definition 1: Task, Experience, Performance

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

- **Task (T)**: what the program is supposed to do. For example, classification (spam vs not spam), regression (predict house prices) or game playing (chess).
- **Experience (E)**: the data the algorithm is exposed to. For example, training set of labeled emails (spam vs ham), past games played by an agent, sensor data from a robot.
- **Performance measure (P)**: the metric used to evaluate progress. For example, classification accuracy (F1 score), mean square error for regression, total reward in reinforcement learning.

A system "learns" if, after seeing more data or interacting more with the environment, its **measured performance improves**.

Example 1: Definition in Action

Some scenarios:

1. **Email Spam Classifier**
   - **T (task)**: Classify emails as spam.
   - **E (experience)**: Training dataset of emails labeled as spam.
   - **P (performance measure)**: Accuracy on unseen emails.

   If accuracy improves as the classifier sees more labeled data, then computer program learning.

2. **Self-Driving Car**
   - **T**: Driving from A to B safely.
   - **E**: Millions of hours of driving footage + sensor readings.
   - **P**: Fewer accidents per mile, shorter trip times.

   If the car improves after more data, it has learned.

[1] Tom Mitchell is a *pioneer of machine learning*, both as a scientist and as an educator. His 1997 textbook, and especially that concise definition, shaped how an entire generation of students and researches understand Machine Learning (ML).

3. **Chess Playing Agent**
   - **T**: Win games.
   - **E**: Past games played against itself or others.
   - **P**: Win rate.

   More games, better play, computer program learning.

This definition matters because is **broad and general** (covers supervised, unsupervised, and reinforcement learning), it stresses **measurable improvement** (no improvement, no learning), and highlights the **central role of data** (E) and evaluation (P).

Why Mitchell's definition doesn't mentions "Machine Learning" explicitly

1. **It's meant to be general**. Mitchell wasn't defining *what ML is as a field*, but rather *what it means for a program to learn*. He avoided vague terms like "machine learning" or "artificial intelligence" and instead described the *process*:
   - A program improves at a **Task (T)**;
   - Thanks to **Experience (E)**;
   - As measured by **Performance (P)**.
2. **Machine Learning = building such programs**. So instead of saying "*Machine Learning is when...*", he framed it as: "*a computer program is said to learn if...*". That's why his definition became the **canonical operational definition of Machine Learning**.
3. **It links directly to practice**. The definition is testable: we can check if a system improves with experience. This is much stronger than a philosophical definition like "*machine learning is making computers intelligent*".

Example 2: Analogy

Think of physics. Newton didn't define "physics". He defined *laws of motion* and *gravity*. From those definitions, physics as a discipline could build itself consistently.
Similarly, Mitchell didn't define "Machine Learning" as a whole discipline. He defined **what it means for a program to learn**. The field then said: "Machine Learning is the study of programs that satisfy this definition".

Mitchell's definition tells us ML is **not about hardcoding solutions**, but about **improving performance with data-driven experience**, measurable by a task-specific metric.

Why we start with Tom Mitchell's definition

1. **Machine Learning is broad and fuzzy**. People use "learning", "AI", "intelligence" loosely. By giving a **formal, authoritative definition** at the beginning, the course sets a *clear baseline*: what do we mean by *learning*? How do we recognize it in a program?

2. **It frames the whole course**. Everything we explain later, supervised learning, neural networks, deep learning, must fit inside this triplet (Task, Experience, Performance). For example:

   - Neural Network training? It's about improving P on T given more E.
   - Reinforcement learning? Same template, different E and P.

3. **It's rigorous but simple**. Unlike philosophical definitions of intelligente, Mitchell's version is **operational**: it tells us *how to test if learning is happening*. It works as a **scientific foundation**, "*if we can't measure performance improvement, we can't claim the program learned*".

4. **It avoids confusion later**. If we started with supervised learning or deep learning right away, we'd lack the general umbrella. With this definition first, we can always check: "*what is our T? what is our E? what is our P?*".

Mathematical View

Formally, suppose we have:

- Dataset $D = \{(x_i, t_i)\}_{i=1}^{N}$ (inputs + targets).
- A model $f_\theta(x)$ with parameters θ.
- A loss function $L(f_\theta(x), t)$ that measures errors (P).

Learning means finding θ^* that minimizes the expected loss:

$$\theta^* = \arg\min_{\theta} \mathbb{E}_{(x,t)\sim E} \left[L\left(f_\theta(x), t \right) \right]$$

This equation will be explained more thoroughly in the following sections.

1.1.1 Machine Learning Paradigms

When Tom Mitchell gave us the **triplet (T, E, P)**, he provided a general definition of learning. But in practice, machine learning problems usually fall into a few **big paradigms**; categories defined by *what kind of data (experience) we provide* and *what kind of task we want solved*. These paradigms are like **different ways of framing the learning problem**:

1. **Supervised Learning**: we give the algorithm examples of input and desired output. The goal is learn to map new inputs to outputs.
2. **Unsupervised Learning**: we only give input data, no labels. The goal is discover hidden structures or representations.
3. **Reinforcement Learning**: we don't provide explicit labels. The system interacts with an environment, receives **rewards or penalties**, and learns a strategy (policy) to maximize reward over time.

These paradigms are important because are the **building blocks of the field**. Almost any ML problem can be described belonging to (or combining) these three. They differ mainly in the **nature of the data (E)** and the **type of feedback (P)** available. Understanding them helps in choosing the right algorithms and models for a problem.

Example 3: Analogy

Imagine teaching three kinds of students:

- **Supervised Learning student**: we show them math problems *with answers*, and they learn how to solve similar ones.
- **Unsupervised Learning student**: we give them a pile of problems *without answers*, and they try to find patterns (like grouping similar problems together).
- **Reinforcement Learning student**: we give them a puzzle game. They don't know the rules, but they learn through *trial and error* because we give them rewards when they succeed.

1.1.1.1 Supervised Learning

Supervised Learning is like learning *with a teacher*:

- The algorithm is given **examples of inputs and their correct outputs (labels)**.
- The goal is to learn a **mapping function** that predicts the correct output for new, unseen inputs.

Formally:

- Training dataset:

$$D = \{(x_1, t_1), (x_2, t_2), \ldots, (x_N, t_N)\}$$

  Where x_i are inputs and t_i are targets.

- Model: $f_\theta(x) \approx t$.
- Learning: choose parameters θ that minimize a loss function measuring error.

In other words, **Supervised Learning** is a type of machine learning where the algorithm is trained on a labeled dataset, meaning each training example includes both the input data and the correct output. And the goal is to learn a function that maps inputs to outputs, in order to make predictions on new, unseen data.

Types of Supervised Learning

In supervised learning we always have:

- **Inputs** x (features).
- **Outputs** t (labels/targets).
- A **model** $f_\theta(x)$ that learns a mapping from inputs to outputs.

The distinction between **classification** and **regression** depends on the **nature of the output**.

- **Classification**: Predict a **discrete class label**. The output space is a finite set of categories. For example:
  - Binary: $\{0, 1\}$, e.g. spam vs not spam.
  - Multi-class: $\{1, \ldots, K\}$, e.g. digits 0-9.

  From a mathematical point of view:

$$f_\theta(x) : \mathcal{X} \to \{1, 2, \ldots, K\}$$

Example 4: Cars vs Motorcycles

Use the classic triplet:

- **Task (T)**: distinguish between two categories (binary classification).
- **Experience (E)**: dataset of images labeled "car" or "motorcycle".
- **Performance (P)**: accuracy (percentage of correct predictions).

Pipeline (how supervised learning was traditionally done before deep learning):

- **Feature Extraction (Hand-Crafted Features)**. Raw data (like an image, sound, or text) is often too complex to give directly to a simple model. Traditionally, humans designed *rules* or *functions* to extract **features** from raw data.
  - Example (images): count edges, corners, textures, or wheel shapes.
  - Example (text): word frequencies, presence of certain keywords.
  - Example (audio): pitch, energy, Mel-frequency coefficients (MFCCs).

  These features are **manually engineered** to capture the most important aspects of the problem. The output is a vector of numbers (feature vector) that represents each example. This step is about "*what information to feed into the model*".

  In this example, hand-crafted features are:

  - Extract "number of circular shapes" (wheels);
  - Extract "dominant color";
  - Extract "edge orientation histograms".

  The photo is now a vector like $[2, 0.6, 0.8]$
- **Learning a Model (Classifier)**. Once we have feature vectors, we train a **machine learning model** that learns to map those features to outputs (labels or numbers). The model **learns decision boundaries** (for classification) or **functions** (for regression) that separate categories or fit numeric values. This is the **actual learning step**: the algorithm adjusts its parameters from the data.

  In this example, the classifier could be a Support Vector Machine (SVM) model, which learns as follows: if "number of wheels ≈ 2" then is a motorcycle; if "number of wheels ≈ 4" then is a car.

- **Regression**: Predict a **continuous value**. The output space is the set of real numbers ($\mathbb{R}$). From a mathematical point of view:

$$f_\theta(x) : \mathcal{X} \to \mathbb{R}$$

Example 5: Price Prediction

Use the classic triplet:

- **Task (T)**: predict a **continuous value** instead of a discrete label.
- **Experience (E)**: dataset of houses (features: size, location, rooms) with their selling prices.
- **Performance (P)**: Mean Squared Error (MSE), Mean Absolute Error (MAE), or R^2 score.

Pipeline:

- **Hand-crafted features**: e.g., number of rooms, square meters, distance to city center.
- **Learned regressor**: a model that predicts a continuous output.

In simple terms, if our labels are:

- Categories, it's **classification**.
- Numbers, it's **regression**.

Why Deep Learning Changed This

In **deep learning**, feature extraction and learning are **not separated anymore**. Neural networks **learn features automatically from raw data** (pixels, sound waves, text). So the pipeline becomes **one end-to-end step**: input raw data → neural network → prediction.

More resources about Supervised Learning can be found in the notes for the Applied Statistics course:

1.1.1.2 Unsupervised Learning

Unsupervised Learning is like learning *without a teacher*:

- We only provide the algorithm with **inputs** $x_1, x_2, \ldots, x_N$.
- There are **no labels/targets** telling the algorithm the "correct answer".
- The goal is to **discover hidden structures** or **representations** in the data.

Formally:

- Dataset:
$$D = \{x_1, x_2, \ldots, x_N\}, \quad x_i \in \mathbb{R}^d$$
- Task: find structure in D, e.g., groups, manifolds, lower-dimension embeddings.
- Performance measure: less obvious (since no labels). It can be internal measures (compact clusters, variance explained) or extrinsic measures (utility in downstream tasks).

The most intuitive unsupervised task: Clustering

In supervised learning, we had "car vs motorcycle", categories are known. In unsupervised, no labels are given. The simplest question becomes: "*can we group the data into natural categories, even if we don't know their names?*". That's exactly what clustering does. **Clustering** is the process of grouping data points into **clusters** such that:

- Points in the same cluster are **similar** to each other.
- Points in different clusters are **dissimilar**.

Clustering uses a **similarity measure**, such as Euclidean distance. The algorithm groups data into clusters that minimize within-cluster distance and maximize between-cluster distance. Some common algorithms include:

- **Hierarchical Clustering**. Build a tree of clusters by progressively merging or splitting. Exists two approach: Agglomerative Clustering (Bottom-Up) or Divisive Clustering (Top-Down).

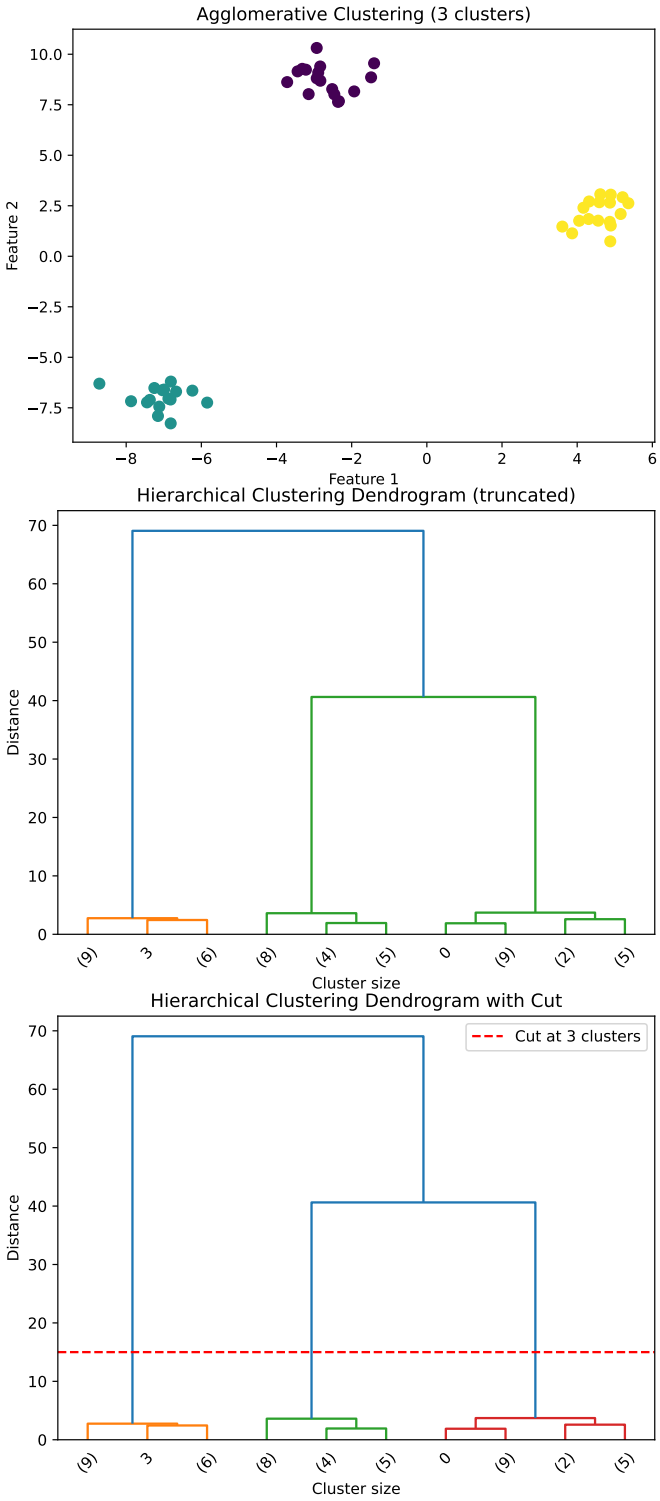


Figure 1: Agglomerative Clustering (top plot), Dendogram (mid plot) and Dendogram with cut (bottom plot).

About Figure 1, page 12. In the Agglomerative Clustering result, each dot is a **data point** (here we generated 50 synthetic points). The algorithm grouped them into **3 clusters**. We can see points within each cluster are **close together** in space. Also, the clusters are **well separated**, this is why hierarchical clustering works well here. The Dendogram shows the **hierarchical merging process**:

- At the **bottom**, each point starts as its own cluster.
- Going **upwards**, clusters that are close together are merged.
- The **height of each merge** (y-axis = distance) indicates how far apart the clusters were when merged.
- At the **top**, all points are eventually merged into a single cluster.

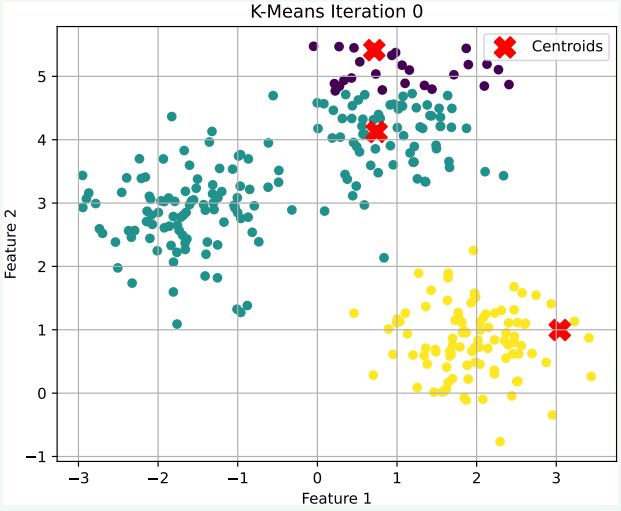
In the last figure, we "cut" the dendogram horizontally at a certain height (distance threshold), and we obtain a chosen number of clusters (here, 3). Everything **below the line** remains as separate clusters. Everything **above the line** (higher merges) is ignored. In the Dendogram, cutting at ≈ 15 gives **3 vertical "branches" crossing the red line**. Each branch corresponds to one cluster. These branches include **all 3 groups of points**.

- **K-Means**. Choose k clusters; assign points to the nearest cluster centroid; and update centroids until convergence.

Example 6: K-Means, taken from the Applied Statistics course

Below is a simple run of the K-means algorithm on a random dataset.

- Iteration 0 - **Initialization**

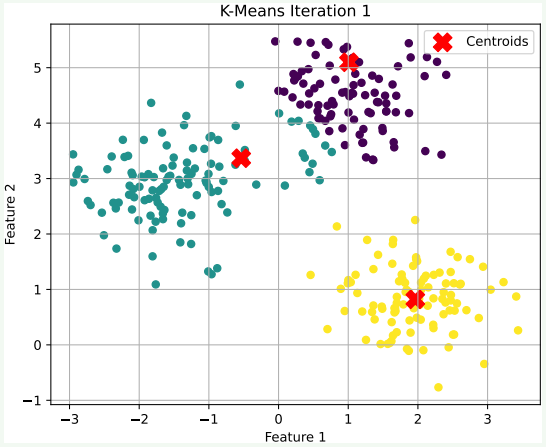


This is the starting point of the K-Means algorithm. **Three centroids are randomly placed in the feature space**.

At this point, no data points are assigned to clusters yet, or all are assumed to be uncolored/unclustered. The positions of the centroids will strongly influence how the algorithm proceeds.

The goal here is to start with some guesses. The next step will use these centroids to form the initial clusters.

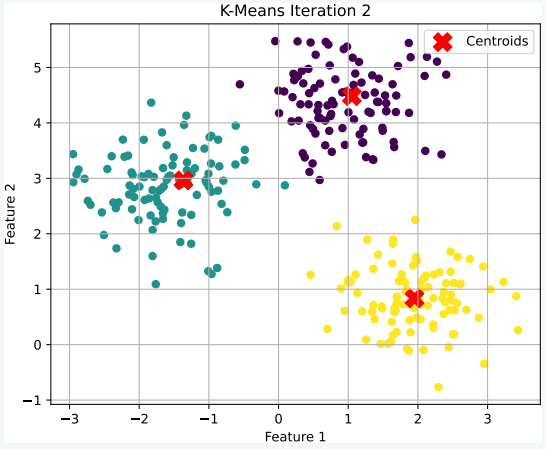
– Iteration 1 - **First Assignment and Update**



Each data point is assigned to the closest centroid, forming the first version of the clusters. New centroids are computed by taking the average of the points in each cluster. We can already see structure forming in the data, as points begin grouping around centroids.

This step is the first real clustering, and centroids begin to move toward dense regions of data.

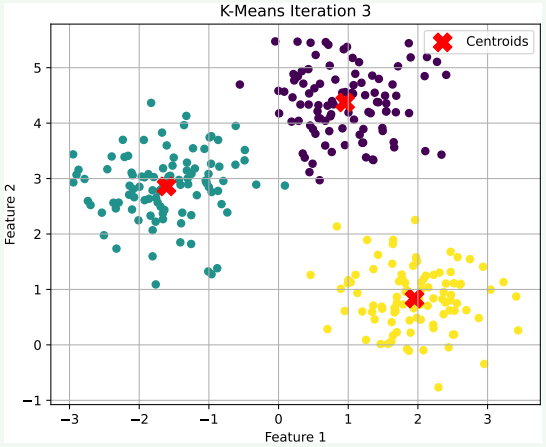
– Iteration 2 - **Re-Assignment and Refinement**



Clusters are recomputed based on updated centroids. Many points remain in the same clusters, but some may shift to a new cluster if a centroid has moved. Centroids continue moving closer to the center of their respective groups.

The algorithm is now refining the clusters and reducing the total distance from points to centroids.

– Iteration 3 - **Further Convergence**



At iteration 3, the K-Means algorithm reached convergence. The centroids no longer moved, and no points changed cluster.This means:

* The algorithm has found a locally optimal solution.
* Further iterations would not improve or change the clustering.
* The final configuration is considered the result of the algorithm.

In practice, this is how K-Means stops: it checks whether the centroids remain unchanged, and if so, it terminates automatically.

More resources about Unsupervised Learning and Clustering can be found in the notes for the Applied Statistics course:

1.1.1.3 Reinforcement Learning

Reinforcement Learning (RL) is like *learning by trial and error*. An **agent** interacts with an **environment** by taking **actions** and receiving **rewards** or **punishments**. The goal of the agent is to learn a policy that maximizes the cumulative reward over time.

At each step, the agent:

1. **Observes a state** s_t from the environment.
2. **Selects an action** a_t based on its current policy $\pi(a_t \mid s_t)$.
3. **Receives a reward** r_t and a **new state** s_{t+1}.

The agent's goal is to learn a **policy** $\pi(a \mid s)$ that maximizes the expected cumulative reward. Unlike supervised learning, no teacher gives the right answer; the agent learns from the **consequences** of its actions.

What is an Agent?

An **agent** is an *entity* that **makes decisions and takes actions in an environment to achieve a specific goal**. In reinforcement learning, the agent learns to optimize its behavior based on feedback from the environment.

With ***entity***, we mean anything that can perceive its environment through sensors and act upon that environment through actuators.

Example 7: Robot Navigation

For example, consider a robot navigating a maze. The robot (agent) perceives its surroundings (state), decides to move left or right (action), and receives feedback (reward) based on whether it gets closer to the exit or hits a wall. The robot's goal is to learn a strategy (policy) that maximizes its chances of reaching the exit while avoiding obstacles.

In simple terms, the robot through cameras and sensors perceives the maze (environment), decides its next move (action), and learns from the outcomes (rewards) to improve its navigation strategy (policy).

In summary:

- **Agent**: The robot.
- **Environment**: The maze.
- **State**: The robot's current position in the maze.
- **Action**: Moving left, right, forward, or backward.
- **Reward**: Positive reward for reaching the exit, negative reward for hitting a wall.
- **Policy**: The strategy the robot uses to decide its next move based on its current state.

The agent's **primary objective** is to **learn a policy that maximizes the cumulative reward** it receives over time by interacting with the environment.

Formalization of Reinforcement Learning

Reinforcement learning problems are often modeled using **Markov Decision Processes (MDPs)**. An MDP is defined by:

- **Task (T)**: learn a policy $\pi(a \mid s)$ mapping states to actions. In other words, the task is to find the best action to take in each state to maximize cumulative reward.
- **Experience (E)**: consists of sequences of states, actions, and rewards obtained by interacting with the environment.
- **Performance Measure (P)**: expected return (sum of discounted rewards):

$$P = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$$

  Where $\gamma \in [0, 1]$ is the discount factor that determines the importance of future rewards.

Key Concepts in Reinforcement Learning

The goal of this section is to introduce the Reinforcement Learning paradigm and its key concepts. These concepts will be covered in more detail in later sections. However, here are some of those concepts:

- **Exploration vs. Exploitation**: The dilemma of choosing between exploring new actions to discover their effects (***exploration***) and exploiting known actions that yield high rewards (***exploitation***).

  Why a dilemma? Because if the agent only exploits known actions, it may miss out on potentially better actions. Conversely, if it only explores, it may not accumulate enough reward.

- **Reward Signal**: The feedback received from the environment after taking an action, used to evaluate the action's effectiveness. It could be sparse or dense:
  - **Sparse Reward**: Rewards are infrequent, making it challenging for the agent to learn. For example, in a game, the agent might only receive a reward upon winning or losing.
  - **Dense Reward**: Rewards are given frequently, providing more immediate feedback. For example, in a driving simulation, the agent might receive small rewards for staying on the road and penalties for going off-road.

- **Delayed reward**: The reward for an action may not be immediate, making it challenging to associate actions with their long-term consequences. For example, in a chess game, a move may not yield an immediate reward but could lead to a win several moves later. The agent must learn to evaluate actions based on their long-term impact rather than immediate outcomes. This requires the agent to consider future rewards when making decisions.

⚖ RL vs. Supervised Learning

Reinforcement learning differs from supervised learning in several key ways:

Aspect	Supervised Learning	Reinforcement Learning
Data	Fixed labeled dataset (in-out pairs)	No labels; agent generates data by acting
Feedback	Correct answer for each example	Rewards (possibly delayed, sparse)
Goal	Minimize error (classification/regression)	Maximize cumulative reward
Typical methods	Regression, SVM, Neural Nets	Q-learning, Policy Gradients, Actor-Critic

⚠ Challenges of Reinforcement Learning

Reinforcement learning presents several challenges:

- **Exploration**: need to try enough actions to discover good strategies.
- **Delayed Feedback**: rewards may not be immediate, complicating reward assignment.
- **Sample inefficiency**: often requires millions of trials to learn effective policies.
- **Stability**: training can be unstable with neural nets.

Despite these challenges, RL has achieved remarkable success in various domains, including game playing, robotics, and autonomous systems.

In summary, reinforcement learning is a powerful paradigm for training agents to **make decisions in complex environments** by **learning from the consequences** of their actions. RL is distinct from supervised learning in its approach to data, feedback, and goals, making it suitable for a wide range of applications where direct supervision is not feasible.

1.2 Towards Deep Learning

This course, and this notes, focuses **mostly on Supervised Learning**, with some unsupervised learning concepts and techniques. ***Why?***

- Supervised Learning is the most widely used paradigm in practice (e.g., image classification, speech recognition, etc.);
- Many deep learning application (image recognition, NLP, etc.) are supervised tasks;
- Unsupervised learning will be touched when needed (e.g., representation learning, generative models, etc.);

Deep Learning is not a new paradigm, it's a **new approach** with supervised/unsupervised learning.

What about Deep Learning? Iris Flower Example

The Iris flower dataset is a classic dataset in machine learning, often used for classification tasks. It consists of 150 samples of iris flowers, each with four features: sepal length, sepal width, petal length, and petal width. The goal is to classify the flowers into three species: Iris setosa, Iris versicolor, and Iris virginica.

- **Traditional Machine Learning Approach:**
  - Extract "good features" from the raw data (e.g., petal length and width);
  - Train a classifier (e.g., decision tree) on these features;
- **Deep Learning Approach:**
  - Learn both **features** and **classifier** simultaneously from the raw data;

For example:

1. If **features are simple** (e.g., petal length and sepal width), then the classification task is **easy**, and a simple model (e.g., decision tree) can achieve high accuracy;
2. If **features are complex** (e.g., raw pixel values of flower images), then the classification task is **hard**, and the traditional approach **struggles to extract meaningful features**;
3. If **impossible to know** which features matter, then handcrafted features are **not enough**, and we need a model that can **learn features** from the data itself (e.g., a deep neural network).
4. Deep Learning learns features **directly from raw data**, making it suitable for complex tasks where feature engineering is challenging or infeasible (hierarchical representations).

⚖ Feature Engineering vs. Learned Features

- **Feature Engineering (Traditional ML)**:
  - Feature Engineering is the process of **using domain knowledge to extract features from raw data** that make machine learning algorithms work. It needs human experts to design and select features that are relevant to the task.
  - **Problem:** requires domain expertise, time-consuming, and may not capture all relevant information. It is often brittle and not transferable to new tasks or domains.
- **Learned Features (Deep Learning)**:
  - Learned Features are features that are **automatically learned by the model** from the raw data during training.
  - Layers learn progressively:
    - Lower layers learn simple patterns (e.g., edges, corners);
    - Middle layers learn more complex patterns (e.g., eyes, wheels);
    - Higher layers learn high-level concepts (e.g., faces, cars).
  - **Advantage:** optimized for the task at hand, can capture complex patterns, and are transferable to new tasks or domains. It requires less manual effort and often generalizes better to unseen data.

1.3 Modern Pattern Recognition (Pre-DL)

Before the rise of deep learning, modern pattern recognition techniques were primarily based on traditional machine learning algorithms and statistical methods. These techniques focused on feature extraction, dimensionality reduction, and classification using various algorithms.

Speech Recognition (early 1990s-2011)

Speech recognition systems used a **multi-stage pipeline** approach, which included:

- **Low-level features**: extracted from the raw audio waveform, such as MFCCs (Mel-Frequency Cepstral Coefficients), a compact representation of the spectral properties of the audio signal.
- **Mid-level features**: built by grouping/encoding low-level features over short time windows, capturing temporal dynamics. For example Mixture of Gaussians (MoG) used to model acoustic units (phonemes).
- **Classifier (high-level features)**: used to map mid-level features to words or phrases. Common classifiers included Hidden Markov Models (HMMs) combined with Gaussian Mixture Models (GMMs) to decode sequences of acoustic units into words.

This pipeline worked decently but was very **hand-crafted** and success depended heavily on the quality of feature engineering.

Object Recognition (2006-2012)

Computer vision systems followed a similar multi-stage pipeline approach:

- **Low-level features**: detect edges, corners, gradients using methods like SIFT (Scale-Invariant Feature Transform) or HOG (Histogram of Oriented Gradients).
- **Mid-level features**: combine low-level descriptors into higher-level "visual words". For example, clustering with k-means to create a codebook of visual words, and Sparse Coding to represent images as sparse combinations of these words.
- **Classifier (high-level features)**: train SVMs (Support Vector Machines) or Random Forests to classify images based on mid-level features.

Again, this approach was heavily reliant on hand-crafted features and required significant domain expertise to design effective features. However, before 2012, these methods were the state-of-the-art in many computer vision tasks.

General Pipeline (Pre-DL Pattern Recognition)

The general pattern recognition pipeline before deep learning can be summarized as follows:

1. **Low-level features**: raw signal transformation (e.g., edges, frequencies).

2. **Mid-level features**: encode or cluster low-level descriptors (e.g., visual words, acoustic units).
3. **Classifier (high-level features)**: learns categories from hand-designed representations.

⚠ Limitations

- **Domain expertise required**: Designing MFCCs, SIFT, HOG, etc. required significant knowledge of the specific domain (speech, vision).
- **task specific**: features built for one task often did not generalize well to others (e.g., MFCCs don't work well for images).
- **Brittleness**: sensitive to noise, illumination, scaling, speaker accents, etc.
- **Limited expressiveness**: as dataset grew, hand-crafted pipelines saturated in accuracy.

Before deep learning, pattern recognition was a **multi-stage pipeline** heavily **reliant on hand-crafted features and domain expertise**. While effective for its time, it had significant limitations in scalability, generalization, and robustness that deep learning would later address.

1.4 What is Deep Learning after all?

After showing the historical context, what Machine Learning is, the three paradigms and how pre-DL pattern recognition worked, we can finally answer the question:

Now that we know what ML does, what makes Deep Learning *different* from classic ML?

We will take our time answering this question. First, we need to understand the meanings of "features" and "classifiers".

What are "features"?

Features are **numerical representations of the raw data** that capture something meaningful for the task.

Type of Data	Raw data example	Example of features
Images	Pixels (RGB values)	Edges, corners, textures
Audio	Waveform (amplitude over time)	Pitch, frequency spectrum, MFCCs
Text	Words or sentences	Word counts, syntactic structure

In **classical ML**, these features were **manually designed** by humans; engineers decided *what* was important and *how to compute it*. For example:

Input image → extract edges manually → feed into SVM classifier

So we had:

Handcrafted Features → Learned Classifier

Where "handcrafted" means "coded by humans". So, before Deep Learning, the **feature extraction** and the **classifier** were two separate stages in the pipeline, and humans designed the first stage. This approach worked, but only if the human correctly guessed *what features matter* for the task.

What does "Learned Features" mean?

Deep Learning says: "*Stop handcrafting features; let the machine learn them automatically, layer by layer, together with the final classifier*".

In **Deep Learning**, the model itself learns how to transform raw data into useful internal representations. Each layer of a neural network acts as a **feature extractor** that learns automatically *what patterns matter*:

- First layers: detect edges, colors, or simple shapes.
- Intermediate layers: detect object parts (e.g., eyes, wheels, leaves).
- Deep layers: detect abstract categories (e.g., "face", "car", "flower").

So instead of telling the machine *what to look for*, we let it **discover patterns directly from data**. This is the "**learned features**" part.

What does "Learned Classifier" mean?

After features are extracted (automatically or manually), the model still needs to **make a decision**: classify, predict, or generate.

- In traditional ML, this is the final **classifier** stage (e.g., SVM, logistic regression, random forest).
- In Deep Learning, the **last few layers** of the network act as that classifier, they map high-level learned features to output labels.

So, both parts, the *feature extractor* and the *decision function*, are **learned jointly** through backpropagation.

$$\text{Raw Data} \xrightarrow[\text{learned weights}]{\text{Feature Extractor}} \text{Representations} \xrightarrow[\text{learned weights}]{\text{Classifier}} \text{Predictions}$$

So DL uses a single model to learn both **features** and **classifier** together: Learned Features + Learned Classifier. The model not only learns *how to decide* but also *how to see the world*, both are learned from data.

So, "What is Deep Learning after all?"

Deep Learning is **not just a new algorithm**, it's a new way of *approaching representation learning*. If we had to answer in one line: "Deep Learning is the automatic learning of hierarchical data representations and decision functions directly from raw data". That's why it's so powerful: it *adapts* to the data and the task, without relying on human intuition about features.

Deepening: Why Not Everything Is Deep Learning

Deep Learning is *powerful*, but it's not a silver bullet, it's not *free*. It's the best tool **when** we have: large amounts of diverse data, high compute, a task based on perception or pattern recognition. Otherwise, **traditional ML or statistical models** can be simpler, faster, and just as effective.

- **Deep Learning needs a *lot of data.*** Deep models have **millions (sometimes billions)** of parameters. They only generalize well when trained on **massive labeled datasets** (e.g., ImageNet: 14M images). If we have small data, like 300 samples from an industrial machine, a deep model will likely **overfit** and perform worse than simpler methods. In other words, Deep Learning shines when there is **data abundance**, but struggles in **data scarcity**.
- **Deep Learning needs a *lot of computation.*** Training is computationally heavy, requiring specialized hardware: GPUs, TPUs, clusters, or cloud computing. Classic ML (SVMs, Decision Trees, Random Forests) can run on a laptop. Deep nets require weeks of GPU training, hyperparameter tuning, and energy cost. So, if the

task doesn't justify the cost, simpler ML is more efficient.

- **Deep models are *black boxes.*** We can rarely explain *why* a deep network made a decision. For critical systems (healthcare, law, finance, safety) we need **interpretability** and **traceability**. Simpler models like linear regression or decision trees are **transparent**, easy to justify in front of regulators or domain experts. For example, a hospital won't risk a deep net saying "tumor" without being able to explain which features caused that prediction.
- **Deep models are *hard to train and tune.*** Choosing architecture (layers, neurons, learning rate, dropout, etc.) is an art. Training can **diverge** or **get stuck** (vanishing gradients, overfitting, exploding losses). We often need extensive experimentation and deep knowledge of optimization tricks. So, not every team or project can afford the expertise and trial cycles DL requires.
- **Deep Learning doesn't always fit the problem.** Some tasks simply:
  1. Have **structured or tabular data** (e.g., bank records, tabular logs). Here, traditional ML (XGBoost, Random Forests) often outperforms DL.
  2. Require **symbolic reasoning** or **logic**, not pattern recognition. Here, DL struggles to capture rules and relationships that classical AI or rule-based systems handle better.
  3. Need **causal inference**, not just correlations. DL finds patterns but doesn't understand cause-effect relationships, which are crucial in many scientific and policy domains. Let's think about a real-world example: predicting disease spread based on interventions (lockdowns, vaccinations) requires understanding causality, not just correlations in data (not just "if X happens, Y follows", but "if we do X, Y will change").
- **Deep Learning needs *good data.*** DL is extremely sensitive to: label noise (wrong annotations ruin learning); biases in the dataset (can reproduce or amplify them); distribution shifts (fails badly if test data differ from training). Traditional methods often handle noise and small variations more robustly. So, "Garbage in $\rightarrow$ garbage out" is even more true with DL.
- **Deep Learning doesn't mean *understanding.*** DL recognizes **patterns**, not meaning. It can detect a cat, but it doesn't *know* what a cat is. It can predict outcomes, but not always *why* they happen. That's why current research explores **hybrid systems** combining DL with: symbolic reasoning (neuro-symbolic AI), knowledge graphs, logic and interpretability layers.

Deepening: ChatGPT, LLaMA & Modern AI Models - What Are They?

ChatGPT, LLaMA, Gemini, Claude, etc. are all based on a specific kind of **Deep Neural Network** called a **Transformer**, introduced in 2017 by Vaswani et al. ("Attention is All You Need"). So, fundamentally:

$$\text{ChatGPT, LLaMA, Gemini, etc.} \in \text{Deep Learning}$$

They are not "beyond" DL, they are its **current frontier**.

What kind of Deep Learning model? They belong to the family of **Large Language Models (LLMs)**.

- **Architecture**: Transformer (a type of deep neural network specialized for sequences and attention).
- **Learning paradigm**: mainly *self-supervised learning*, a subform of unsupervised learning.
- **Objective**: predict the next word (token) given the previous ones.

Mathematically:

$$P\left(w_t \mid w_1, w_2, \ldots, w_{t-1}\right)$$

"Given this context, what's the next most probable word?". That's the only thing it learns. Everything else (reasoning, style, facts) *emerges* from learning this next-token distribution on vast text corpora.

Why are they still called "Deep Learning"? They perfectly fit the definition we discussed earlier: "Deep Learning is the learning data representation and decision functions directly from data".

- They learn **representations** of words, sentences, and even concepts automatically.
- They have **layers upon layers** (up to 100+ in GPT-4).
- They **don't rely on hand-crafted linguistic features** (no human tells them grammar rules).
- They learn everything **directly from raw text data** (syntax, semantics, even reasoning patterns).

So they exemplify:

$$\begin{gathered}\text{Learned Features (embeddings)} + \\ \text{Learned Classifier (next word predictor)}\end{gathered}$$

But at **massive scale**, with **billions of parameters** and trained on **terabytes of text**. This scale is what enables their surprising capabilities.

What makes them *different* from earlier Deep Learning. Traditional DL (e.g., CNNs, RNNs) had strong **task specialization**: CNNs for vision, RNNs for sequences, LSTMs for time series. Instead, Transformers with LLMs changed the game because they are **general-purpose learners**:

- They can handle language, code, images, audio, even multimodal data.
- Their **attention mechanism** learns relationships between all parts of the input simultaneously.

They are sometimes called: "Foundation Models", because they can be *fine-tuned* for many downstream tasks (translation, summarization, question answering, etc.).

Why do they feel intelligent? When we train on *massive data* (trillions of words) and *huge models* (hundreds of billions of parameters), the model starts showing **emergent behaviors**:

- Understanding context, humor, and nuance.
- Performing reasoning and arithmetic.
- Generating coherent, creative text.
- Translating languages fluently.
- Writing code snippets.

But still, it's pattern prediction. There is **no explicit symbolic reasoning** or understanding; it's just learned statistical structure at enormous scale. So we say: "They are **Deep Learning models**, trained on **massive dataset**, showing **emergent intelligence**".

1.5 What's Behind Deep Learning?

If the concept of neural networks exists since the 1950s, ***why did Deep Learning explode only after 2012?*** This is a natural question that comes *after* we've seen what Deep Learning is. To answer this question, we show two perspectives: the **MIT view** and **The Economist view**.

The MIT view: Computational Power

According to MIT and many early researchers, Deep Learning became possible only when **computational resources** caught up with the theory. It means that the mathematics and algorithms (backpropagation, perceptrons, convolutional nets) existed for decades, but **training deep networks** requires enormous computation:

- Millions of matrix multiplications.
- Thousands of gradient updates per sample.
- Gigantic datasets.

Before 2010, this was impractical. Around 2011-2012, **GPUs** (Graphics Processing Units) changed everything:

- They made large-scale matrix computations thousands of times faster.
- Deep learning frameworks (Theano, TensorFlow, PyTorch) made GPU computing accessible.
- Hardware parallelism allowed training networks with **hundreds of layers** instead of 3-4.

So from the MIT perspective: Deep Learning rose because **we finally had the computational power to train deep models**.

The Economist view: Big Data

In 2012, *The Economist* (yes, the famous magazine) proposed a different, and equally valid, explanation: "Deep Learning exploded because the world finally generated **enough data** to feed it". It means that the Internet, social media, smartphones, sensors, and cloud storage created **massive labeled datasets**:

- ImageNet (over 14 million labeled images).
- YouTube (millions of labeled videos).
- Text from web, Wikipedia, books, perfect for LLM pretraining.

Deep neural networks thrive on data volume: they don't generalize well with few examples. The more data, the better they learn **hierarchical representations**. So from the Economist perspective: "Deep Learning rose because **we finally had Big Data**, the fuel it needs to work".

The Real View: Both Matter

In reality, both perspectives are correct and complementary. Deep Learning's success is due to the **synergy of computational power and big data**:

- Before 2010, algorithms existed but computing was too slow and data too scarce. Then neural networks were limited to shallow architectures and small datasets.
- Around 2012, hardware (GPUs, TPUs, distributed training) made computation feasible. Simultaneously, the explosion of digital data provided the massive labeled datasets needed.

This combination triggered the **Deep Learning revolution**. The turning point was **ImageNet 2012**, where Krizhevsky, Sutskever, and Hinton demonstrated that a deep convolutional network (AlexNet) could drastically outperform traditional methods on image classification. This success was possible only because:

- They used two NVIDIA GPUs to train a deep network with millions of parameters.
- They trained on the large ImageNet dataset with 1.2M labeled images.

The result was an error rate of 15%, compared to 26% for the best traditional method. This landmark event showcased the **power of deep learning when both computational resources and big data are available**.

1.6 Summary

Everything we've seen, supervised, unsupervised, or reinforcement learning, ultimately depends on **how we represent data**. In traditional ML, features are *hand-crafted*. In Deep Learning, features are *learned automatically* through hierarchical representations. The revolution of Deep Learning wasn't new math, it was learning **what matters** in the data instead of coding it by hand.

$$\text{Success of ML} \Rightarrow \text{Success of its feature representation}$$

Deep networks just made the **representation learning** automatic and scalable.

Deep Learning = Learning Data Representation from Data

Deep Learning is not a specific architecture (like CNN, RNN, or Transformers) or algorithm. It's the **paradigm** where:

1. Input → raw data (e.g., pixels, text, audio)
2. Model → multiple non-linear layers learning internal representations.
3. Output → desired prediction/task.
4. Learning → end-to-end optimization of all layers together.

So instead of:

$$\text{Human designs features} \rightarrow \text{Model learns mapping}$$

We now have:

$$\text{Model learns both features and mapping} \rightarrow \text{directly from data}$$

This is the essence of Deep Learning: **learning hierarchical representations directly from raw data**.

"Which data?" - The key question of the course

This is the **transition line** to the rest of the course (notes). Now that we know *what* Deep Learning is, the next question is *what data we use and how*. Different data types define the upcoming sections:

Data Type	Upcoming Section
Tabular / numerical	Perceptrons & Feed-Forward NNs
Images	Convolutional Neural Networks (CNNs)
Sequential (text, time series)	Recurrent Neural Networks (RNNs) & Transformers
Unlabeled data	Autoencoders & Word Embeddings

So this question of "which data?" becomes the **roadmap** for the rest of the course (notes).

2 From Perceptrons to FNNs

2.1 Historical Context

When Artificial Intelligence first emerged as a field in the 1940s and 1950s, researchers were fascinated by the idea of creating machines that could *think*, *adapt*, and *learn* as the human brain does. At that time, traditional computers were already capable of executing precise, deterministic instructions with incredible speed. However, these **early machines lacked flexibility**: they could not interpret noisy or ambiguous input, nor could they modify their behavior from experience.

This limitation led scientists to look beyond the rigid Von Neumann architecture[2] and toward the **brain** as an alternative computational paradigm. The human brain, with its billions of interconnected neurons, represented a radically different kind of machine: **massively parallel**, **distributed**, **redundant**, and **fault-tolerant**. Each neuron is *simple*, yet together they form a system capable of extraordinary complexity and adaptability.

From this inspiration arose the idea of **neural networks**: mathematical models built from simple interconnected units that imitate, in a highly abstract way, the behavior of biological neurons. Interestingly, neural networks are not a recent invention of the deep learning era: they have existed since the birth of AI itself. In fact, the phrase "*Deep Learning is not AI, nor Machine Learning*" emphasizes that **deep learning is a later evolution within this larger historical continuum**. Neural networks have been a foundational approach to artificial intelligence from its inception, long before modern computational power and data made them successful.

In summary, the reason researchers in the 1940s and 1950s looked "beyond Von Neumann" was that they sought to create machines that could **learn from experience** and **adapt to new situations**, capabilities that traditional computers lacked:

- **1940s motivation**: classic computers excelled at precise, fast arithmetic but researches wanted systems that could **interact with noisy data**, be **parallel and fault-tolerant**, and **adapt**.
- **Brain as a computational model**: the brain offers a radically different architecture that is massively parallel, distributed, redundant system. These properties are an appealing template for computation, which inspired artificial neurons and later full neural networks.

The inception of AI

In the years immediately following the Second World War, a new scientific dream began to take shape: the idea that intelligence could be recreated in a machine. Early pioneers such as **Alan Turing**, **John von Neumann**, **Warren McCulloch**, and **Walter Pitts** laid the foundations of what would soon

[2] The sequential model where computation and memory are separated

be called *Artificial Intelligence*. Computers had just proven they could follow precise instructions and perform huge calculations at incredible speed, yet these machines were nothing more than rigid automata: they obeyed every command literally, unable to perceive, reason, or learn.

The emerging field of AI was born from the desire to bridge that gap, to make machines that could **adapt**, **generalize from experience**, and **interact intelligently** with the world. The 1940s and 1950s were therefore an era of conceptual excitement: *could the brain's mechanisms be modeled mathematically and implemented in hardware or software?* The earliest experiments sought to replicate the nervous system's structure, creating computational units that mimicked neurons and synapses. These units could, in principle, activate or remain silent depending on the inputs they received, a primitive form of reasoning.

At this stage, AI and neural networks were inseparable: **to build an intelligent machine meant to build an artificial brain**. Over the next decades, this vision would split into two main traditions. One emphasized *symbolic* reasoning (manipulating explicit rules and logic) while the other, the *connectionist* approach, pursued learning from examples through networks of simple computational nodes. The second line, though overshadowed for many years, would eventually resurface as what we now call **Deep Learning**.

From Von Neumann Machines to Brain-Inspired Models

In the 1940s, the **Von Neumann architecture** defined what we still call a *classical computer*: a machine with a central processor (CPU) that executes instructions stored in memory, step by step, following a deterministic sequence. This design is extremely powerful for arithmetic and logic, but it has key limitations when the goal is to emulate intelligence.

A Von Neumann computer is **serial**, **rigid**, and **exact**: it does exactly what it's told, line by line. Intelligence, however, requires something different, the ability to handle **noisy or incomplete data**, **recover from errors**, **adapt to change**, and **operate in parallel** on many signals at once. The human brain, in contrast, is a **massively parallel** and **distributed** system made of roughly 10^{11} neurons, each connected to thousands of others through 10^{14} to 10^{15} synapses.

This comparison motivated the idea of a **computational model inspired by the brain**. Instead of a single central processor, the brain uses huge numbers of simple processing units (neurons) working together. Each neuron performs a small, nonlinear operation, but their collective behavior gives rise to perception, reasoning, and learning.

Researchers realized that if intelligence in humans comes from these interactions, perhaps machines could become intelligent by simulating **networks of artificial neurons**, each following simple rules, but collectively capable of complex, adaptive computation.

In short:

- Von Neumann: deterministic, sequential, rigid.
- Brain-inspired: parallel, adaptive, fault-tolerant.

This shift marks the conceptual birth of **neural networks** as a new computational paradigm.

Neural Networks in the Early AI Era

The idea of taking the **human brain** as a model for computation stems from its extraordinary complexity and efficiency. A typical adult brain contains around **100 billion neurons** (10^{11}), and each neuron is connected to roughly **7'000** others, forming an estimated 10^{14} - 5×10^{14} **synapses**, even reaching 10^{15} in a three-year-old child.

Despite being slow compared to digital processors (neurons fire in milliseconds, not nanoseconds), the brain's power lies in its **massive parallelism** and **redundancy**. Each **neuron is a simple processing element**, but together they create a distributed, nonlinear, and fault-tolerant system capable of perception, reasoning, adaptation, and learning; functions that no single algorithmic machine of the 1940s could perform.

From a computational viewpoint, this means:

- **Processing is distributed**: no central control; intelligence arises from interactions.
- **Information is encoded collectively**: a concept survives even if some neurons fail.
- **Parallelism ensures speed and robustness**: thousands of operations occur simultaneously.
- **Adaptivity**: synaptic strengths (connections) change with experience, enabling learning.

These characteristics inspired the **first attempts to formalize "neurons" mathematically**, giving rise to the **perceptron** and to the field of *artificial neural networks*. The perceptron is, in essence, a simplified abstraction of how a biological neuron integrates inputs, applies a threshold, and produces an output. An idea that we'll explore in the following section.

⚖ What about the computation of biological versus artificial neurons?

🧠 In a **biological neuron**, information is transmitted through **electrochemical signals**:

- The **dendrites** receive inputs from other neurons through *synapses*.
- Each input can be **excitatory** (it increases activation) or **inhibitory** (it decreases activation).
- The neuron **integrates** all these signals in the **cell body (soma)**.
- When the total accumulated signal exceeds a **threshold**, the neuron **fires**, sending an output through its **axon** to other neurons.

Although this process is complex and involves various biochemical mechanisms, it can be summarized as:

$$\text{collect inputs} \rightarrow \text{integrate} \rightarrow \text{compare with threshold} \rightarrow \text{fire}$$

⚙ But how to model this computationally? In the **artificial version**, we simplify this biological process into a mathematical model:

$$h_j(x, w, b) = f\left(\sum_{i=1}^{I} w_i x_i - b\right) = f\left(w^T x\right)$$

Where:

- x_i are the input values (analogous to signals received by dendrites). They are like the neurotransmitter signals that a biological neuron receives from other neurons.
- w_i are the weights (analogous to synaptic strengths). They represent how strongly each input influences the neuron's activation.
- b is the bias (analogous to the threshold). It determines the level of input required for the neuron to activate.
- $f(\cdot)$ is the activation function (analogous to the firing mechanism). It decides whether the neuron fires based on the integrated input.

Each artificial neuron thus performs three main steps:

1. **Weighted sum** of its inputs (integration): $\sum_{i=1}^{I} w_i x_i$.
2. **Subtracts the bias** (thresholding): $\sum_{i=1}^{I} w_i x_i - b$.
3. **Applies the activation function** (firing decision): $f\left(\sum_{i=1}^{I} w_i x_i - b\right)$.

Definition 1: Artificial Neuron

An **Artificial Neuron** is a **mathematical model** inspired by the way a biological neuron works. It's the **basic computation unit** of a neural network.

While a real neuron collects electrical signals from thousands of connections (synapses) and "fires" if the total signal passes a threshold, an artificial neuron does the same thing, but with numbers.

Formally, it takes several inputs $(x_1, x_2, \ldots, x_I)$, multiplies each by a **weight** w_i, sums them, adds a **bias** b, and passes the result through an **activation function** $f(\cdot)$:

$$h_j(x, w, b) = f\left(\sum_{i=1}^{I} w_i x_i - b\right) = f\left(w^T x\right) \tag{1}$$

Where:

- **Inputs** (x_i): the signals coming from other neurons or from data (e.g., pixel values).
- **Weights** (w_i): how strong each input connection is (analogous to synaptic strength).
- **Bias** (b): shifts the activation threshold up or down.
- **Activation function** (f): decides whether the neuron "fires" (outputs a strong signal) or stays quiet.

In essence, the pipeline of an artificial neuron is:

Weighted sum $\rightarrow$ Threshold/Bias $\rightarrow$ Nonlinear activation $\rightarrow$ Output

Definition 2: Bias

The **Bias** is an additional parameter in an artificial neuron that allows the activation function f to be shifted horizontally, providing the model with the ability to represent patterns that do not pass through the origin.

Mathematically, it appears as the constant term b in the neuron's activation equation (see page 35):

$$a = w^T x + b$$

The bias represents the **intrinsic tendency of a neuron to activate**, even in the absence of input. It acts like a tunable threshold that controls *when* the neuron fires.

Think of the bias as the neuron's **default tendency to fire**, it decides *how easy or hard* it is for the neuron to activate:

- A **large positive bias** → neuron tends to fire even with small input.
- A **large negative bias** → neuron needs strong evidence (large input sum) to fire.

In other words, the bias *shifts the activation threshold* left or right along the input axis, allowing the neuron to learn more complex decision boundaries.

Imagine a simple rule: "*if the weighted sum of our inputs is greater than 0, we output 1*". Now suppose all our inputs are zero ($x_1 = x_2 = 0$). If we want the neuron to still fire in that case, we need a **bias** to "push" it over the threshold. Bias gives the neuron a *baseline activity*, like saying: "*even if there's not input, we are slightly inclined to fire*".

So, an **artificial neuron** mimics the logical essence of a biological one: a small computing unit that combines multiple inputs into one output, depending on the learned connection strengths (weights) and a bias term. This is the foundation of the **perceptron**, the first neural network model, the topic of the next section.

2.2 The Perceptron

2.2.1 Who Invented It?

Once researchers realized that the brain could be viewed as a network of simple processing units, the next natural step was to formalize this idea into an actual **computational model**, what we now call a **neural network**.

Definition 3: Neural Network

A **Neural Network** is simply a **collection of artificial neurons** (page 35) connected by weighted links. Each neuron:

- Receives inputs,
- Computes a weighted sum,
- Applies an activation function,
- And produces an output that becomes the input for the next neuron.

Through these connections, the network forms a structure capable of **transforming input data into meaningful outputs**, a function approximator that *learns* by adjusting its weights.

The very first implementations appeared in the 1940s-1960s, with three major milestones:

McCulloch & Pitts (1943). They proposed the **Threshold Logic Unit (TLU)**, the first mathematical model of a neuron. Each unit:

- Received multiple binary inputs,
- Multiplied them by fixed weights,
- Summed them up,
- Compared the sum to a threshold,
- Output 1 if the threshold was exceeded, 0 otherwise.

They proved that a network of such units could represent **any logical function**, meaning it could, in theory, "compute" anything if properly wired.

Frank Rosenblatt (1957). He built the first **trainable model**, the **Perceptron**. Rosenblatt's perceptron could automatically **learn** the correct weights from examples using an update rule based on errors. His prototype was implemented in hardware:

- The weights were stored as adjustable electrical components (potentiometers),
- Electric motors updated them during learning. This was the first step from theoretical neuroscience to **machine learning**.

Bernard Widrow (1960). He developed the **ADALINE (Adaptive Linear Neuron)** and later the **MADALINE (Multiple ADALINE network)**. Widrow's key idea was to express the threshold as a **bias term**, simplifying the equations and making it easier to train models using gradient-based optimization, a cornerstone of modern networks.

Together, these models represent the **first generation of neural networks**: simple, linear systems inspired by the brain but operating with mathematics and electricity. They laid the groundwork for the more complex architectures that would follow, leading to the deep learning revolution we see today.

2.2.2 Mathematical Model & Logical Operations

The **Perceptron** is the **simplest neural network**, a single neuron that transforms multiple input signals into one output through a weighted sum and a thresholding function.

Formally, given inputs:

$$x = [x_1, x_2, \ldots, x_I]$$

And weights:

$$w = [w_1, w_2, \ldots, w_I]$$

The perceptron computes the quantity:

$$a = \sum_{i=1}^{I} w_i x_i + b = w^T x + b \tag{2}$$

Where:

- x_i are the input features,
- w_i are the learnable connection weights,
- b is the **bias** (representing the firing threshold).

Then, this **activation** a passes through a **step function** (also called **threshold** or **activation function**) to produce the final output y:

$$y = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

In some conventions, the output can also be -1 or $+1$ instead of 0 and 1, depending on how the data is encoded.

Sometimes, we include the bias directly as a weight w_0 associated with a fixed input $x_0 = 1$, rewriting the equations as:

$$y = f\left(w_0 x_0 + w_1 x_1 + \ldots + w_I x_I\right) = f\left(w^T x\right) \tag{4}$$

This makes formulas simpler and more uniform for training algorithms (compact vector notation).

Interpretation of the Perceptron math

The perceptron divides the input space into two regions separated by a **decision boundary** (a hyperplane[3]). If the weighted sum of inputs exceeds the threshold, the neuron "fires" (outputs 1); otherwise, it stays silent (outputs 0). Thus,

[3] A **hyperplane** is a **generalization of a line or a plane** to any number of dimensions. It's the mathematical way to describe a *flat surface* that separates space into two parts. In 1D, a hyperplane is just a point that splits the line into two halves; in 2D, it's a line that divides the plane into two regions, one where the perceptron outputs 1 and the other where it outputs 0; in 3D, it's a plane that separates space into two halves. In higher dimensions, it remains a flat subspace that partitions the input space.

the perceptron acts as a **linear classifier**: it determines which side of the hyperplane the input vector lies on.

We can express the **exact set of points where the neuron is undecided** (the **Decision Boundary Equation**) by setting the activation a to zero:

$$w^T x + b = 0 \quad \text{(decision boundary equation)} \tag{5}$$

What it can actually do: Logical Operations

Now, if the perceptron is a computational unit, *what kind of computations can it perform?* To answer that, we need simple, well-defined **functions** to test it on. The most basic functions are the **logical operations** used in Boolean algebra. Logical operations (like AND, OR, NOT) are perfect because:

- They have **binary inputs** (0 or 1), exactly like neuron activations.
- They produce **binary outputs** (true or false), like the perceptron's step function.
- They let us see immediately whether the neuron can separate input cases correctly.

Logical operations are the **first experiments** that show the perceptron's power as a *linear classifier*.

When the perceptron can reproduce logic operations like AND or OR, it proves that:

1. A single neuron can implement **decision-making**.
2. The model is capable of **classification** (separating inputs into categories).
3. We can assign **geometric meaning** (a hyperplane dividing true/false examples).

Example 1: Logical OR ($\vee$)

x_1	x_2	$y = x_1 \vee x_2$
0	0	0
0	1	1
1	0	1
1	1	1

We want the perceptron to output **1** if *any* input is 1. A possible set of parameters is:

$$w_1 = 1, \quad w_2 = 1, \quad b = -0.5$$

This gives us the activation function:

$$a = w_1 x_1 + w_2 x_2 + b = x_1 + x_2 - 0.5$$

Or equivalently:

$$y = \begin{cases} 1 & \text{if } x_1 + x_2 - 0.5 > 0 \\ 0 & \text{otherwise} \end{cases}$$

So each neuron computes:

$$y = f\left(w_1 x_1 + w_2 x_2 + b\right) = f\left(1 \cdot x_1 + 1 \cdot x_2 - 0.5\right)$$

Checking all input combinations:

- For $(0, 0)$: $a = 0 + 0 - 0.5 = -0.5 \Rightarrow y = 0$
- For $(0, 1)$: $a = 0 + 1 - 0.5 = 0.5 \Rightarrow y = 1$
- For $(1, 0)$: $a = 1 + 0 - 0.5 = 0.5 \Rightarrow y = 1$
- For $(1, 1)$: $a = 1 + 1 - 0.5 = 1.5 \Rightarrow y = 1$

Thus, the perceptron correctly implements the OR function.

Example 2: Logical AND ($\wedge$)

x_1	x_2	$y = x_1 \wedge x_2$
0	0	0
0	1	0
1	0	0
1	1	1

We want the perceptron to output **1** only if *both* inputs are 1. A possible set of parameters is:

$$w_1 = 1, \quad w_2 = 1, \quad b = -1.5$$

This gives us the activation function:

$$a = w_1 x_1 + w_2 x_2 + b = x_1 + x_2 - 1.5$$

Or equivalently:

$$y = \begin{cases} 1 & \text{if } x_1 + x_2 - 1.5 > 0 \\ 0 & \text{otherwise} \end{cases}$$

So each neuron computes:

$$y = f\left(w_1 x_1 + w_2 x_2 + b\right) = f\left(1 \cdot x_1 + 1 \cdot x_2 - 1.5\right)$$

Checking all input combinations:

- For $(0, 0)$: $a = 0 + 0 - 1.5 = -1.5 \Rightarrow y = 0$
- For $(0, 1)$: $a = 0 + 1 - 1.5 = -0.5 \Rightarrow y = 0$
- For $(1, 0)$: $a = 1 + 0 - 1.5 = -0.5 \Rightarrow y = 0$

- For $(1, 1)$: $a = 1 + 1 - 1.5 = 0.5 \Rightarrow y = 1$

Thus, the perceptron correctly implements the AND function. However, we can see that other weight/bias combinations could achieve the same result. For example:

$$w_1 = 1.5, \quad w_2 = 1.5, \quad b = -2.0$$

In both examples, the perceptron defines a **line (in 2D)** that separates input combinations giving output 1 from those giving output 0. For OR, the line lies closer to the origin, since only $(0, 0)$ should give 0; for AND, the line lies further away, since only $(1, 1)$ should give 1. So, by adjusting weights and bias, the perceptron can learn to classify inputs according to these logical rules. However, it's clear that **manually setting weights and biases for complex tasks is impractical**. This brings us to the next important topic: ***how can it learn those weights automatically instead of us setting them by hand?***

2.2.3 Hebbian Learning Rule

Now that we understand what the Perceptron does and who invented it, let's explore **how it learns** from data. When the first artificial neurons were proposed, researchers wanted them not just to compute, but to **learn from experience**, as biological neurons do. The earliest and most influential idea for this was the **Hebbian Learning Rule**, introduced by psychologist **Donald Hebb** in 1949.

The biological intuition

Donald Hebb was a psychologist, not a mathematician. In 1949, he was trying to explain **how the brain learns from experience**, without having explicit "teachers" or formulas. He observed that, in biological brains, learning seems to happen **through association**. That's the origin of his famous sentence:

"*Cells that fire together, wire together.*"

This means that if **two neurons** are **active at the same time** (one sending a signal and the other firing) then the **connection** (synapse) between them should **become stronger**. Over time, the brain reinforces useful associations automatically.

In other words, if neuron A consistently helps activate neuron B, the connection from A to B should be strengthened. This principle is thought to underlie learning and memory formation in the brain.

The Artificial Version: Mathematical Formulation

Now, we translate this biological intuition into a mathematical rule that can be applied to the Perceptron. In artificial neurons, "firing" means *output* is active (e.g., output is 1). So if both input and output are active at the same time, that's equivalent to "they fired together". The Hebbian learning rule says:

- **Increase** the weight of connections that are active when the neuron fires.
- **Decrease** or leave unchanged the connections that are inactive or misaligned.

To translate this into a mathematical rule for a Perceptron, we **express the weight update** as follows:

$$\Delta w_i = \eta \cdot x_i \cdot t \tag{6}$$

- If $x_i > 0$ (input is active) and $t > 0$ (target output is active), both are active, then Δw_i is positive, so the weight w_i **increases**, the connection strengthens.
- If $x_i > 0$ (input is active) but $t \leq 0$ (target output is inactive), mismatch, then Δw_i is zero or negative, so the weight w_i **decreases** or remains the same, the connection weakens.
- If $x_i \leq 0$ (input is inactive), regardless of t, then no update occurs since Δw_i is zero, the connection remains unchanged.

Where:

- η is the **learning rate**, a small positive constant that controls how much the weights are adjusted during each update. It ensures that learning is gradual and stable. To make an analogy, think of η as the **speed limit** on a road: it prevents the learning process from speeding ahead too quickly and potentially crashing (i.e., diverging).
- x_i is the i^{th} **input value** to the Perceptron.
- t is the **target output** (desired response) for the given input.
- Δw_i is the **change in weight** for the i^{th} input. This change is added to the current weight w_i to get the new weight.

The full update rule becomes:

$$w_i^{(k+1)} = w_i^{(k)} + \Delta w_i = w_i^{(k)} + \eta \cdot x_i \cdot t \tag{7}$$

This is the **Weight Update Rule**. It tells us *how to modify* each connection w_i after seeing one training example. Conceptually, at each learning step (each training example):

1. **Take the current weights** $w_i^{(k)}$.
2. **Compute** how much they should change $\Delta w_i = \eta \cdot x_i \cdot t$.
3. **Add that change** to get the new weights $w_i^{(k+1)}$.

How it works

1. **Initialize** all weights w_i to small random values (or zeros).
2. **Set** the learning rate η to a small positive value (e.g., 0.01).
3. For each **training example** (x, t):
   - **Compute the Perceptron's output** y using the current weights:
   $$y = f\left(w^T x\right)$$
   - **Compare with the target** t.
     - If $y = t$, the output y matches the target t, the neuron is already correct, so **no weight update is needed** since the association is already learned.
     - If $y \neq t$, the output y does not match the target t, the neuron is incorrect, and we need to **update the weights** to strengthen the association. This is done using the Hebbian learning rule:
     $$w_i^{(k+1)} = w_i^{(k)} + \eta \cdot x_i \cdot t$$
     This can be explained in informal steps:
     * For each weight w_i, compute the change $\Delta w_i = \eta \cdot x_i \cdot t$
     * Update the weight: $w_i \leftarrow w_i + \Delta w_i$
4. Repeat until all examples are correctly classified or a stopping criterion is met (e.g., a maximum number of iterations).

Example 3: Hebbian Learning Rule

Let's say we're learning a simple OR function with two inputs x_1 and x_2. The target outputs t for the four possible input combinations are:

- $x = [0, 0] \rightarrow t = 0$
- $x = [1, 0] \rightarrow t = 1$
- $x = [0, 1] \rightarrow t = 1$
- $x = [1, 1] \rightarrow t = 1$

We do not include a bias term in this example for simplicity. We'll use a step activation function:

$$f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

The algorithm proceeds as follows:

1. **Initialize weights** $w_1 = 0.0$, $w_2 = 0.0$ and learning rate $\eta = 0.1$.
2. **First training example** $x = [0, 0]$, $t = 0$:
   - **Compute output**: $y = f(0.0 \cdot 0 + 0.0 \cdot 0) = f(0) = 0$
   - Output matches target, so **no weight update needed**.
3. **Second training example** $x = [0, 1]$, $t = 1$:
   - **Compute output**: $y = f(0.0 \cdot 0 + 0.0 \cdot 1) = f(0) = 0$
   - Output does not match target, so we **update weights**:

$$\begin{aligned} \Delta w_1 &= 0.1 \cdot 0 \cdot 1 = 0.0 \\ \Delta w_2 &= 0.1 \cdot 1 \cdot 1 = 0.1 \\ w_1 &\leftarrow 0.0 + 0.0 = 0.0 \\ w_2 &\leftarrow 0.0 + 0.1 = 0.1 \end{aligned}$$

4. **Third training example** $x = [1, 0]$, $t = 1$:
   - **Compute output**: $y = f(0.0 \cdot 1 + 0.1 \cdot 0) = f(0) = 0$
   - Output does not match target, so we **update weights**:

$$\begin{aligned} \Delta w_1 &= 0.1 \cdot 1 \cdot 1 = 0.1 \\ \Delta w_2 &= 0.1 \cdot 0 \cdot 1 = 0.0 \\ w_1 &\leftarrow 0.0 + 0.1 = 0.1 \\ w_2 &\leftarrow 0.1 + 0.0 = 0.1 \end{aligned}$$

5. **Fourth training example** $x = [1, 1]$, $t = 1$:
   - **Compute output**: $y = f(0.1 \cdot 1 + 0.1 \cdot 1) = f(0.2) = 1$
   - Output matches target, so **no weight update needed**.

After one pass through the training data, the weights are $w_1 = 0.1$ and $w_2 = 0.1$. Repeating this process over multiple epochs will further refine the weights until the Perceptron correctly models the OR function.

Should the bias be updated if the output doesn't match the target?

In the Hebbian learning rule, the **bias term can also be updated similarly to the weights**. The bias can be treated as a weight connected to an input that is always 1. Therefore, if the output does not match the target, the bias should also be updated to help correct the output. The update rule for the bias b would be:

$$\Delta b = \eta \cdot x_0 \cdot t = \eta \cdot 1 \cdot t = \eta \cdot t \qquad x_0 = 1 \tag{8}$$

So, if the **output** is **incorrect**, the bias would be adjusted by adding Δb to the current bias value:

$$b^{(k+1)} = b^{(k)} + \Delta b = b^{(k)} + \eta \cdot t \tag{9}$$

This adjustment helps shift the activation threshold of the Perceptron, making it more likely to produce the correct output in future iterations.

In other words, we can think of the **bias** as a *special weight* w_0 that connects to a *constant input* $x_0 = 1$. This trick lets us treat the bias **exactly the same** as all the other weights in the update rule. Therefore, the neuron computes:

$$y = f\left(w_0 \cdot 1 + w_1 x_1 + w_2 x_2 + \dots\right)$$

And the update rule applies uniformly to **every** w_i, including w_0 (the bias):

$$w_i^{(k+1)} = w_i^{(k)} + \eta \cdot x_i \cdot t \quad \text{for } i = 0, 1, 2, \dots$$

Thus, at every iteration:

- The **normal weights** $(w_1, w_2, \dots)$ adapt based on the input features and target output.
- The **bias** b (also considered a weight, like w_0) is updated to help the Perceptron better fit the data. It adapts based on the target output t alone, since its associated input is always 1 (i.e., $x_0 = 1$).

The bias learns to **adjust the overall tendency** of the neuron to fire. If the network often needs to output 1 (positive target), the bias weight increases, making it easier for the neuron to activate. Conversely, if the network often needs to output 0 (negative target), the bias weight decreases, making it harder for the neuron to activate. This dynamic adjustment of the bias is crucial for the Perceptron to learn effectively from data.

2.2.4 Perceptron as Linear Classifier

A **classifier** is a model that assigns input data points to one of several classes. In the case of the perceptron, it classifies input vectors into two classes based on a linear decision boundary.

A **linear classifier** is a type of classifier that makes its decisions based on a linear combination of the input features. In poor words, it makes a decision by checking on which side of a *line* (in 2D), *plane* (in 3D), or *hyperplane* (in higher dimensions) the input data point lies.

The perceptron computes:

$$a = w^T x + b$$

where w is the weight vector, x is the input vector, and b is the bias term, and decides:

$$y = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{if } a \leq 0 \end{cases} \tag{10}$$

So the **decision** happens depending on the *sign* of a: positive values lead to class 1, while zero or negative values lead to class 0.

The **Decision Boundary** is the exact set of points where the model is **undecided**, where it switches from one class to the other. That happens precisely when the condition changes sign from negative to positive. The "border" between those two cases is when the activation a equals **zero**. Formally, this occurs when:

$$w^T x + b = 0$$

That's where the perceptron's decision flips, and therefore it's the **boundary line (or hyperplane)**. This boundary divides the input space into two halves:

- Points where $w^T x + b > 0$ are classified as class 1.
- Points where $w^T x + b < 0$ are classified as class 0.
- Points where $w^T x + b = 0$ lie exactly on the decision boundary.

Wait, why is zero special? In the above equation (10), the perceptron outputs 0 when $a = 0$. Why is it called the decision boundary? In theory, the **boundary**:

$$w^T x + b = 0$$

Is **not assigned to any class**, it's the **limit** between them. Exactly on the boundary ($a = 0$), the model *is indifferent*, because **geometrically** that point is the **separator**, not really part of any region (see Figure 2, page 48, to visualize this concept). However, in practice, the $\leq$ sign in the perceptron decision rule is just a **tie-breaking rule**, otherwise we wouldn't know what to output when $a = 0$. But for geometry and theory, we're interested in **where the switch happens**, so we call the exact set of points the **decision boundary**.

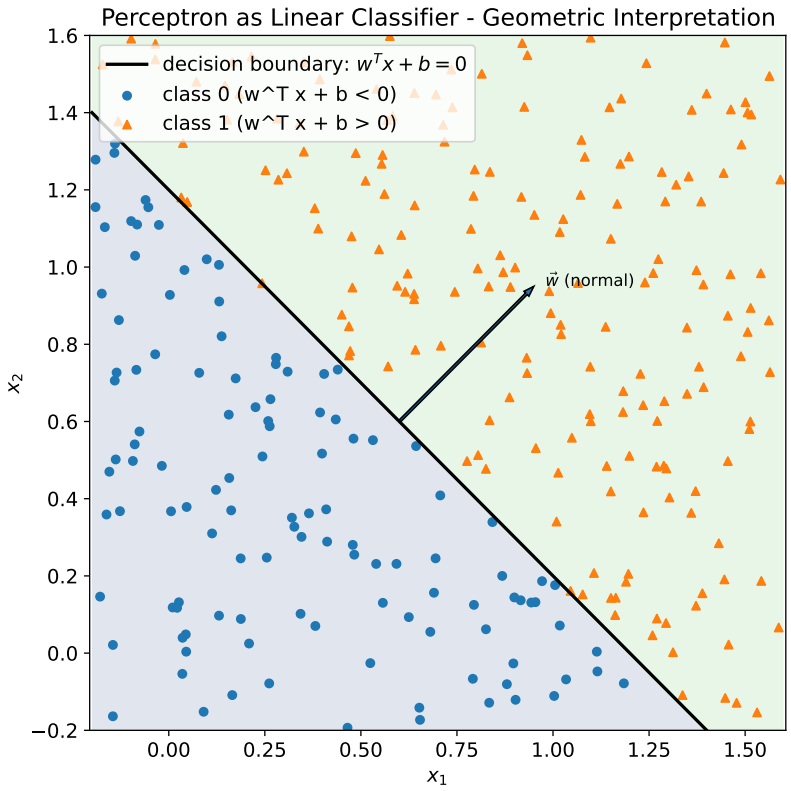


Figure 2: A 2D example of a perceptron as a linear classifier. The line represents the decision boundary where $w^T x + b = 0$. Points on one side of the line are classified as class 1 (green area, orange triangles, everything that satisfies $w^T x + b > 0$), while points on the other side are classified as class 0 (blue, $w^T x + b < 0$). The arrow indicates the **normal vector** $\vec{w}$, which is perpendicular to the decision boundary and points towards the class-1 side. The normal vector $\vec{w}$ points in the direction where the perceptron output increases.

Concept	**Meaning**
w	Defines the *direction* of the separating hyperplane.
b	Shifts the hyperplane from the origin.
$w^T x + b = 0$	Equation of the decision boundary (hyperplane).
$w^T x + b > 0$	Region classified as class 1.
$w^T x + b < 0$	Region classified as class 0.
$\vec{w}$	Normal vector to the decision boundary, indicating the direction of increasing output.
Limitation	Can only classify linearly separable data.

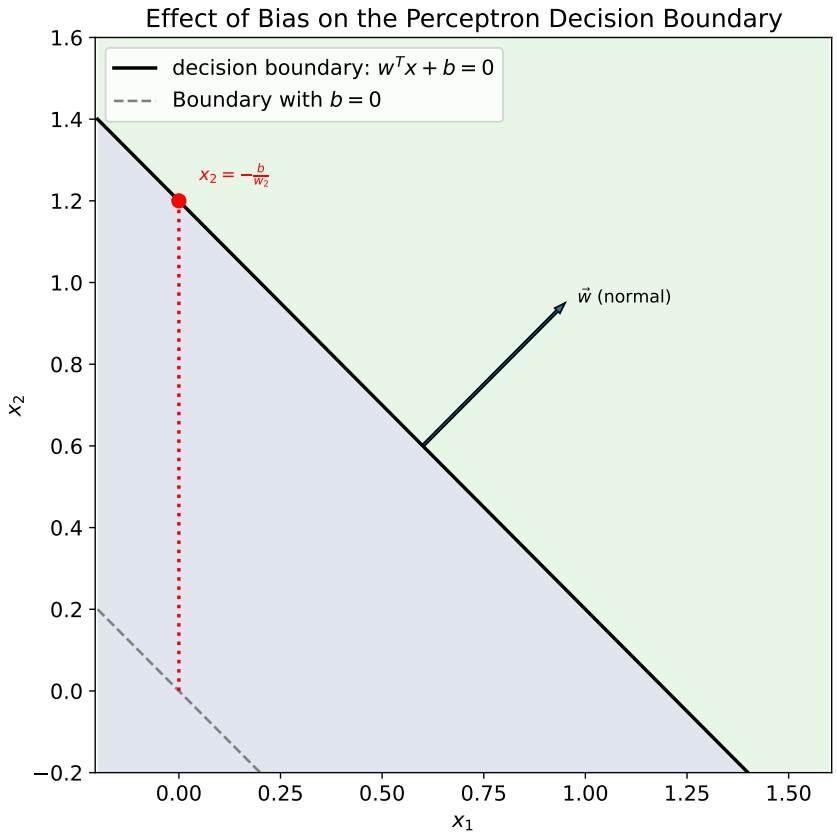


Figure 3: Geometric interpretation of the bias in a perceptron. The solid black line shows the decision boundary $w^T x + b = 0$ for $b = -1.2$, while the dashed gray line represents the case $b = 0$. The red dotted segment highlights the vertical shift of the intercept caused by the bias. The normal vector $\vec{w}$ is perpendicular to the boundary and points toward the region where the neuron output is 1 ($w^T x + b > 0$).

❓ If the bias is negative, why does the boundary shift upwards?
Imagine $w = [1, 1]$. Then $w^T x + b = x_1 + x_2 + b$. Without bias ($b = 0$), the boundary is:

$$x_1 + x_2 = 0$$

Is the **line through the origin** at a 45-degree angle. Now, if we **add** $b = -1.2$, the boundary becomes:

$$x_1 + x_2 - 1.2 = 0 \quad \Rightarrow \quad x_1 + x_2 = 1.2$$

This line is **shifted upwards** because for any given x_1, x_2 must be larger to satisfy the equation. Thus, a **negative bias** shifts the decision boundary **upwards**, while a **positive bias** would shift it **downwards**. In this case, for x_2 direction, the bias effectively **increases** the threshold that x_2 must reach to cross the boundary:

$$x_2 = -x_1 + 1.2$$

2.2.5 Boolean Operators & Linear Separability

Once we've seen that a perceptron can learn **logical functions** (like AND, OR), the next natural question is:

"*Can it learn **all** possible logical operators?*"

Short answer: **No**. And understanding why leads to the crucial idea of **linear separability**: the key limitation of the perceptron model.

Let's summarize the four fundamental binary logical functions (i.e., functions with two binary inputs and one binary output):

Operator	**Output = 1 when...**	**Linearly separable?**
AND	both inputs are 1	✔ Yes
OR	at least one input is 1	✔ Yes
NAND	at least one input is 0	✔ Yes
NOR	both inputs are 0	✔ Yes
XOR	exactly one input is 1	✖ No
XNOR	both inputs are the same	✖ No

Note that the first four operators (AND, OR, NAND, NOR) are all **linearly separable**, while the last two (XOR, XNOR) are **not**. But what does "linearly separable" mean in this context?

The game changer: *Linear Separability*

Definition 4: Linearly Separable

A dataset is **Linearly Separable** if there **exists** a straight line (in 2D), plane (in 3D), or **hyperplane** (in higher dimensions) that **perfectly divides** the **two classes of data points**. That is, all points of one class lie on one side, and all points of the other class lies on the opposite side.

Formally, given a dataset with two classes, it is linearly separable if there exist weights $w_1, w_2, \ldots, w_n$ and a bias b such that for every data point $(x_1, x_2, \ldots, x_n)$:

$$\begin{cases} w_1x_1 + w_2x_2 + \ldots + w_nx_n + b > 0 & \text{if the point belongs to Class 1} \\ w_1x_1 + w_2x_2 + \ldots + w_nx_n + b < 0 & \text{if the point belongs to Class 2} \end{cases}$$

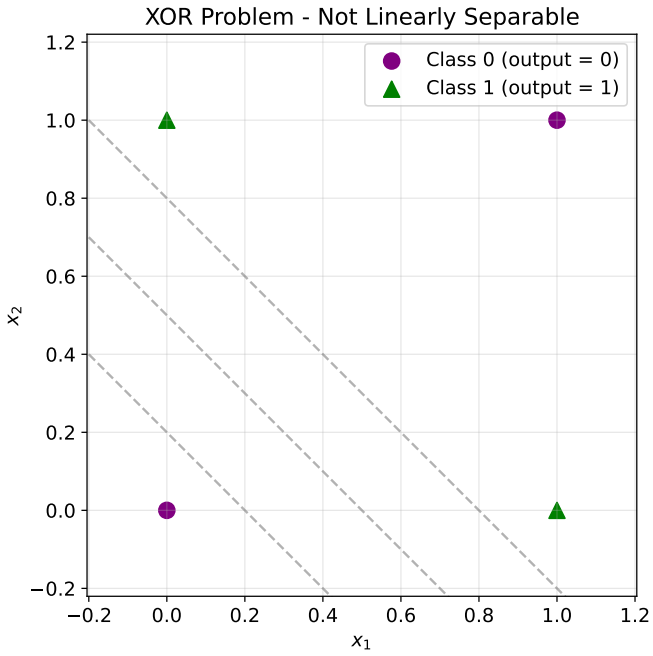
If such weights and bias exist, the dataset is linearly separable. Otherwise, no single perceptron can solve it (i.e., classify it correctly).

⚠ The XOR problem - The classic example of non-linear separability

Until now, we've seen that perceptrons can learn linearly separable functions like AND and OR. However, the linear separability limitation becomes evident when we consider some logical functions, such as XOR (exclusive OR). A little reminder of the XOR truth table:

x_1	x_2	XOR(x_1, x_2)
0	0	0
0	1	1
1	0	1
1	1	0

The XOR function outputs 1 only when exactly one of its inputs is 1. If we plot the input-output pairs of the XOR function on a 2D plane, we get the following points:



Here, the points are arranged in an "X" pattern:

- Class 1 points are at (0, 1) and (1, 0) (opposite corners).
- Class 0 points are at (0, 0) and (1, 1) (remaining corners).

No single straight line can separate the Class 1 points from the Class 0 points. We'd need *two lines* forming a region (a non-linear boundary). Hence, the XOR function is **not linearly separable**, and a single-layer perceptron cannot learn it.

In summary, the perceptron can only create **linear decision boundaries**, so:

- ✅ It perfectly models **linearly separable** problems (like AND, OR, simple threshold rules).
- ❌ If fails for **non-linearly separable** problems (like XOR, parity, circle-vs-ring, etc.).

This realization in the 1960s led to what's often called the "**AI winter**," as researchers recognized the limitations of single-layer perceptrons. However, this challenge also paved the way for the development of **multi-layer neural networks** (and backpropagation), which can overcome these limitations by creating complex, non-linear decision boundaries, combining multiple perceptrons in layers.

2.3 Feed-Forward Neural Networks (FNNs)

2.3.1 Architecture

After discovering that a single perceptron can only draw **one straight boundary**, researchers realized that we need **multiple layers** of neurons to build **non-linear decision surfaces**. That's how **Feed-Forward Neural Networks (FNNs)** were born.

Definition 5: Feed-Forward Neural Networks (FNNs)

A **Feed-Forward Neural Network (FNN)** is an artificial neural network where information **flows in one direction only**, from the input layer, through any hidden layers, to the output layer. There are no cycles or loops in the network.

$$x \rightarrow \text{Layer 1} \rightarrow \text{Layer 2} \rightarrow \ldots \rightarrow \text{Output Layer}$$

Each layer receives signals from the previous layer, processes them using weighted connections and activation functions, and passes the output to the next layer.

Structure of a Feed-Forward Network

An FNN is composed of:

1. **Input Layer**: one neuron per input feature (e.g., pixel value, sensor reading). Does not perform computation, it simply distributes inputs to the next layer.
2. **Hidden Layer(s)**: Contain neurons that each compute:

   $$a_j = f\left(w_j^T x + b_j\right)$$

   where w_j are the weights, b_j is the bias, x is the input vector from the previous layer, and f is the nonlinear activation function (sigmoid, tanh, ReLu, etc.). The index j identifies the specific neuron in the hidden layer. Each neuron learns different **intermediate features** of the data.
3. **Output Layer**: Produces the network's final result. Activation depends on the task, we mean for classification we often use **softmax** or **sigmoid**, while for regression we use a **linear** activation.

The connections between neurons are **weighted**:

- Each neuron in layer l is connected to **all** neurons in the previous layer $l-1$. This is called a **fully connected** or **dense** layer.
- Every connection has its own **weight**, which is learned during training. Every neuron also has a own **bias** term.
- During training, all these weights and biases are adjusted to minimize the difference between the predicted output and the actual target values.

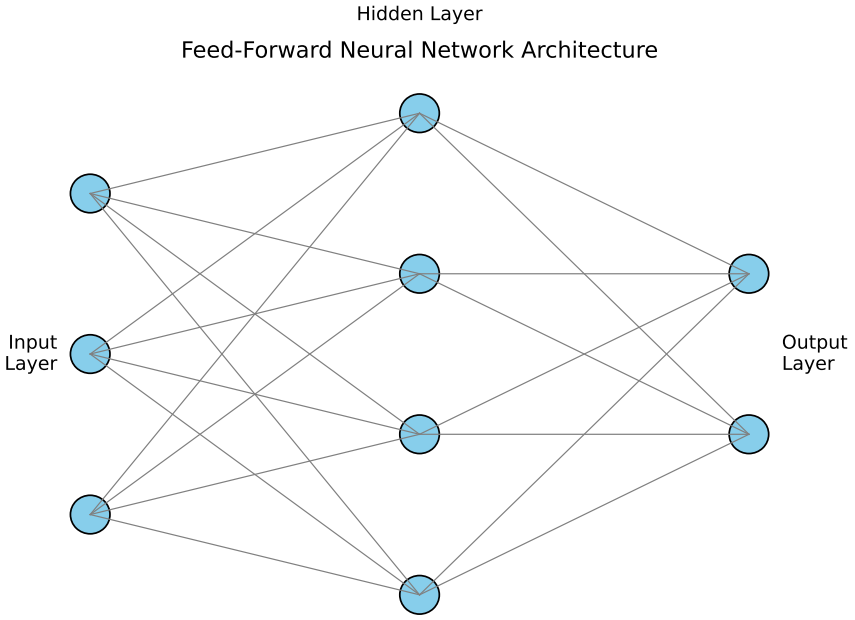


Figure 4: Architecture of a simple feed-forward neural network. Each layer is fully connected to the next one. Signals flow in one direction (input → hidden → output) without feedback connections.

FNNs can be represented as graphs based on this architecture:

- **Nodes** represent neurons.
- **Edges** represent weighted connections between neurons.

This graph representation helps visualize the network's structure and understand how information propagates through it.

Mathematically, for a layer l:

$$\begin{cases} a^{(l)} = f\left(W^{(l)} a^{(l-1)} + b^{(l)}\right) & \text{for hidden layers} \\ x^{(l)} = f\left(a^{(l)}\right) & \text{for output layer} \end{cases}$$

where:

- $a^{(l)}$ is the activation vector of layer l.
- $W^{(l)}$ is the weight matrix connecting layer $l-1$ to layer l.
- $b^{(l)}$ is the bias vector for layer l.
- f is the activation function.
- $x^{(l)}$ is the final output of the network.

This formalism allows us to **compute the output of the network given an input vector by sequentially applying these transformations layer by layer**.

How FNNs learn hierarchical features

Adding layers lets the network learn **hierarchical representations**:

- The first layers capture **simple patterns** (e.g., edges in images).
- Deeper layers combine these simple patterns into **more complex features** (e.g., shapes, objects).

This ability to build abstractions through depth is the essence of **Deep Learning**.

2.3.2 Activation Functions

Every neuron computes a **weighted sum** of its inputs and bias:

$$a = w^T x + b$$

And then applies an **activation function** $f(a)$ to produce its final output:

$$y = f(a)$$

The **Activation Function** defines **how the neuron "fires"**, i.e., how it transforms the raw input signal into an output that will be passed to the next layer. Exist many different activation functions, each with its own characteristics and use cases. The choice of activation function can significantly impact the performance and capabilities of a neural network. In the following, we will explore some of the most commonly used activation functions in neural networks.

In the next sections, we will mention the derivative result and the range of each activation function:

- During training, neural networks learn by **minimizing a loss function**, and this requires **backpropagation**, which is based entirely on **derivatives**. We will explain backpropagation later, but for now, here is a brief overview of how it works: (1) each neuron has parameters w_i (weights) and b (bias); (2) to adjust them, we compute how the **loss** changes if we slightly change each parameter; (3) mathematically, that's done through **gradients**, the derivatives of the loss with respect to the weights. When we apply the **chain rule** to compute these gradients, we get something like:

  $$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial a} \cdot \frac{\partial a}{\partial w_i}$$

  Where $\frac{\partial y}{\partial a}$ is the derivative of the activation function $f(a)$. Therefore, having an activation function: **too small** $(= 0)$, means gradients vanish and the learning stops; **too large** gradients can cause instability. Hence, the derivative of the activation function is crucial for effective learning.

- The **range** of an activation determines what kind of outputs each neuron can produce, and this affects: (1) **how the next layer receives data**, and **how easy it is to train** the network. For example, if an activation function outputs values in a limited range (like between 0 and 1), it can help keep the network's outputs stable and prevent extreme values that could lead to numerical issues during training.

2.3.2.1 Linear

The **Linear Activation Function** is the simplest activation function, defined as:

$$f(a) = a \tag{11}$$

That means the neuron's output equals its input; there's no distortion or thresholding. So the neuron is just a **weighted sum** followed by nothing.

Intuitive interpretation

If all neurons in a network use $f(a) = a$, then every layer just performs a **linear transformation** of the input. Stacking **multiple linear layers doesn't add any expressive power**; the entire **network can be reduced to a single linear transformation**. In general, for a network with n layers, each represented by a weight matrix W_i, the overall transformation is:

$$f\left(W_n\left(W_{n-1}\left(\ldots W_2\left(W_1 x\right)\ldots\right)\right)\right) = \left(W_n W_{n-1}\ldots W_2 W_1\right) x$$

However, if the network only uses linear activation functions, then it simplifies to:

$$f\left(W_2\left(W_1 x\right)\right) = \left(W_2 W_1\right) x$$

Therefore, linear activation functions are rarely used in practice for hidden layers, as they cannot capture complex patterns in data. In other words, a purely linear network cannot learn anything more complex than a straight boundary; it is basically a big matrix multiplication because all the layers collapse into one.

Property	Description
Formula	$f(a) = a$
Derivative	$f'(a) = 1$
Range	$(-\infty, +\infty)$
Nonlinear?	✗
Typical use	Regression output layers (not hidden neurons)

When to use it

Even though a **linear activation** is useless inside hidden layers (because it doesn't add nonlinearity), it's still **important at the output layer** of certain models:

- **Regression problems**: when we want a real-valued output (like predicting house prices), a linear activation allows the network to produce any value in the range $(-\infty, +\infty)$. However, the linear activation is applied only at the output layer, while hidden layers use nonlinear activations to capture complex patterns.
- **Autoencoders or embedding layers**: sometimes the linear activation helps maintain continuous representations of data.

In summary, the **linear activation** keeps the output proportional to the input. It's mathematically simple and differentiable, but **does not allow the network to model nonlinear relationships**. Hence, not used in hidden layers.

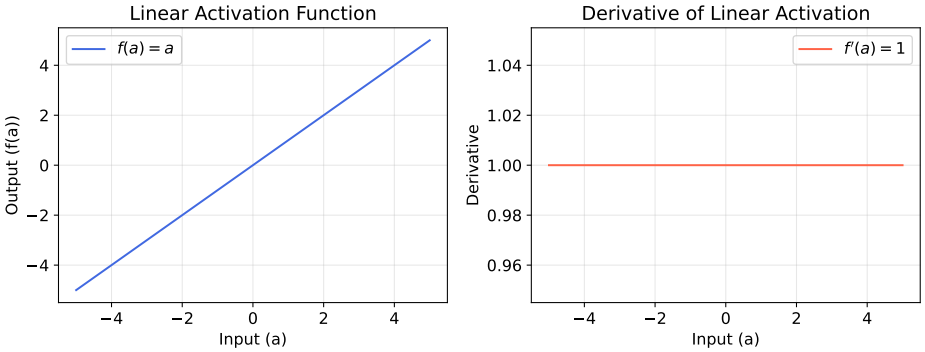


Figure 5: Linear activation $f(a) = a$ and its derivative. The function is the identity (a straight line, showing that the neuron outputs exactly what it receives), and its constant derivative $f'(a) = 1$ allows perfect gradient flow. However, being linear, it adds no expressive power to the network.

2.3.2.2 Sigmoid

The **Sigmoid Activation Function** (or **Logistic Activation Function**) is defined as:

$$f(a) = \frac{1}{1 + e^{-a}} \tag{12}$$

It "**squashes**" any real-valued **input** a **into a range** between **0 and 1**.

The Sigmoid converts its input into something that looks like a **smooth threshold**:

- ⇈ Large positive inputs a produce outputs close to 1;
- ⇊ Large negative inputs a produce outputs close to 0;
- ≈ Inputs a close to 0 produce outputs close to 0.5

It's often described as giving a "**firing probability**" to a neuron, mimicking how biological neurons activate gradually rather than with a hard step.

Graphically, the Sigmoid function is a smooth **S-shaped** curve (sigmoidal). It's **continuous** and **differentiable everywhere**. It has a gentle slope around 0 and saturates near the extremes (0 or 1).

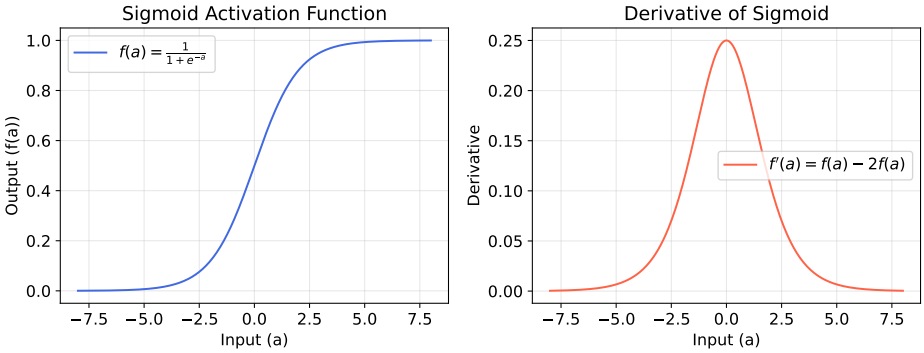


Figure 6: Sigmoid Activation Function and its derivative. The sigmoid introduces smooth nonlinearity and maps inputs into $(0, 1)$, but its derivative vanishes for large inputs, causing slow learning in deep networks.

The **derivative** tells us how sensitive the neuron's output is to changes in its input. For the Sigmoid function, the derivative is given by:

$$f'(a) = f(a) - 2f(a) = f(a) \cdot [1 - f(a)] \tag{13}$$

This means:

- When $f(a) \approx 0.5$, the derivative is maximized at 0.25, allowing for significant weight updates during training (**neuron is responsive**).
- When $f(a) \approx 0$ or $f(a) \approx 1$, the derivative approaches 0, leading to very small weight updates (**neuron is saturated** and gradients vanish).

  This **vanish gradient problem** makes deep networks with Sigmoid activations **hard to train**, as gradients become very small in earlier layers

during backpropagation. In other words, the Sigmoid function can cause **slow learning** in deep networks due to its saturating behavior at extreme input values.

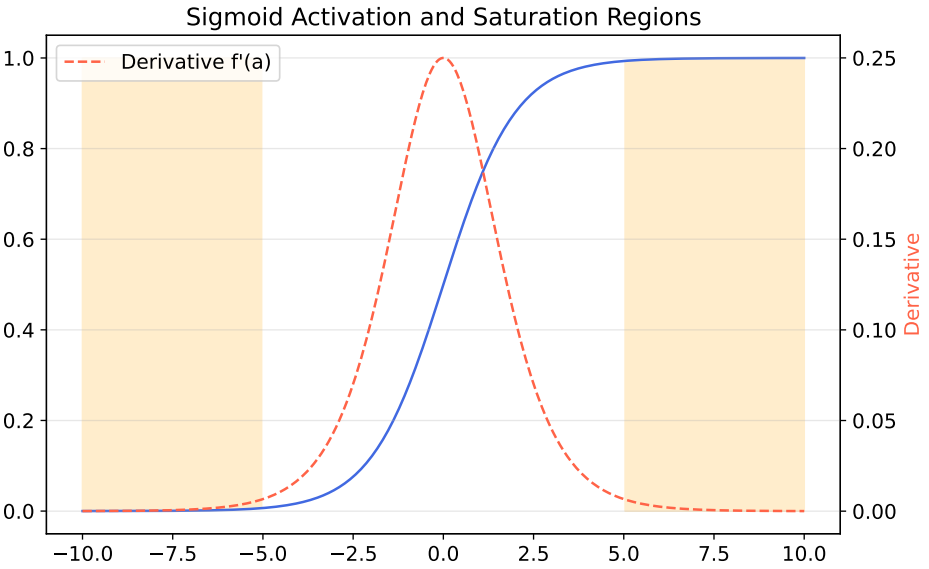


Figure 7: Sigmoid activation and saturation regions.

In figure 7 we can see the **vanish gradient problem** in action: when neurons saturate, their gradients vanish, making it hard for the network to learn from data during training.

- The blue curve shows the **sigmoid activation** $f(a)$. That smooth **S-shaped curve** (in blue) represents:

$$f(a) = \frac{1}{1 + e^{-a}}$$

  In the center (around $a = 0$), the output is about 0.5, and the curve is **steepest**; for large positive $a > 5$, the curve **flattens near 1**; for large negative $a < -5$, it **flattens near 0**. Those flat tails are the **saturation regions** (highlighted in orange). These regions mean that when the neuron receives very strong positive or negative inputs, its output doesn't change much anymore; it has reached its "max" or "min" activation.

- The orange curve are the parts of the curve where the output is **almost constant**:
  - On the left (for large negative a), the output is very close to 0 (saturated low);
  - On the right (for large positive a), the output is very close to 1 (saturated high).

  In those regions:

$$\frac{\partial f}{\partial a} = f'(a) \approx 0$$

So the neuron has **stopped responding**, even big changes in a cause almost no change in the output $f(a)$.

- The red curve shows the **derivative** $f'(a)$:

$$f'(a) = f(a) \cdot (1 - f(a))$$

The derivative is only significant in a **small central region** (roughly between -3 and 3). Outside this range, the derivative **drops to near zero**, indicating that the neuron is **saturated** and **not learning effectively**.

Property	**Value / Meaning**
Formula	$f(a) = \dfrac{1}{1+e^{-a}}$
Range	$(0, 1)$
Derivative	$f'(a) = f(a) \cdot (1 - f(a))$
Output interpretation	Probability or "firing strength".
Pros	Smooth, differentiable, bounded output, probabilistic interpretation.
Cons	Vanishing gradients for large a, outputs not zero-centered, computationally expensive.

When to use it

- **Output layer of binary classification** networks, where outputs represent probabilities:

$$\mathbb{P}(y = 1 \mid \mathbf{x}) = f(w^T \mathbf{x} + b)$$

- Historically used in hidden layers (in early networks), but now often replaced by ReLU or its variants due to vanishing gradient issues.

In summary, the Sigmoid activation function is essentially a **soft version** of the perceptron's step function:

$$\text{step: } f(a) = \begin{cases} 1 & \text{if } a \geq 0 \\ 0 & \text{if } a < 0 \end{cases} \quad \longrightarrow \quad \text{sigmoid: } f(a) = \frac{1}{1+e^{-a}}$$

So the sigmoid allowed neural networks to become **differentiable**, which made **gradient-based learning (backpropagation)** possible. However, its tendency to **saturate** and cause **vanishing gradients** has led to the adoption of alternative activation functions (like ReLU) in modern deep learning architectures.

2.3.2.3 Hyperbolic Tangent (tanh)

The **Hyperbolic Tangent (tanh) Activation Function**, commonly known as **tanh**, is defined as:

$$f(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}} \tag{14}$$

And its derivative is:

$$f'(a) = 1 - \tanh^2(a) = 1 - f^2(a) \tag{15}$$

The tanh function maps input values to an output range between -1 and 1. It is a scaled version of the sigmoid function, centered around zero.

The tanh activation function looks **very similar to the sigmoid**, but it is **symmetric around zero**: outputs range from -1 to 1, which helps in centering the data and can lead to faster convergence during training. This makes it **zero-centered**, which is a big advantage.

- When the input is zero ($a = 0$), the output is also zero ($f(0) = 0$).
- For large positive inputs, the output approaches 1 ($f(a) \to 1$ as $a \to +\infty$).
- For large negative inputs, the output approaches -1 ($f(a) \to -1$ as $a \to -\infty$).

That means hidden neurons can have both positive and negative activations, which helps later layers learn faster because the data stays **balanced** around zero.

Why it's better than sigmoid

- **Range**: The tanh function outputs values between -1 and 1, while the sigmoid function outputs values between 0 and 1. This means that tanh is zero-centered, which can help with convergence during training.
- **Gradient around zero**: The derivative of the tanh function is ≈ 1 around zero, while the derivative of the sigmoid function is ≈ 0.25 around zero. This means that the tanh function has a steeper gradient[4] around zero, which can help with learning.
- **Saturates?** Both functions can saturate for large positive or negative inputs, leading to the vanishing gradient problem. However, because tanh is zero-centered, it can help mitigate this issue to some extent.
- **Training speed**: In practice, models using the tanh activation function often converge faster than those using the sigmoid function, especially in deep networks.

[4] A "steeper gradient" means that small changes in the input lead to larger changes in the output, which can help the model learn more effectively.

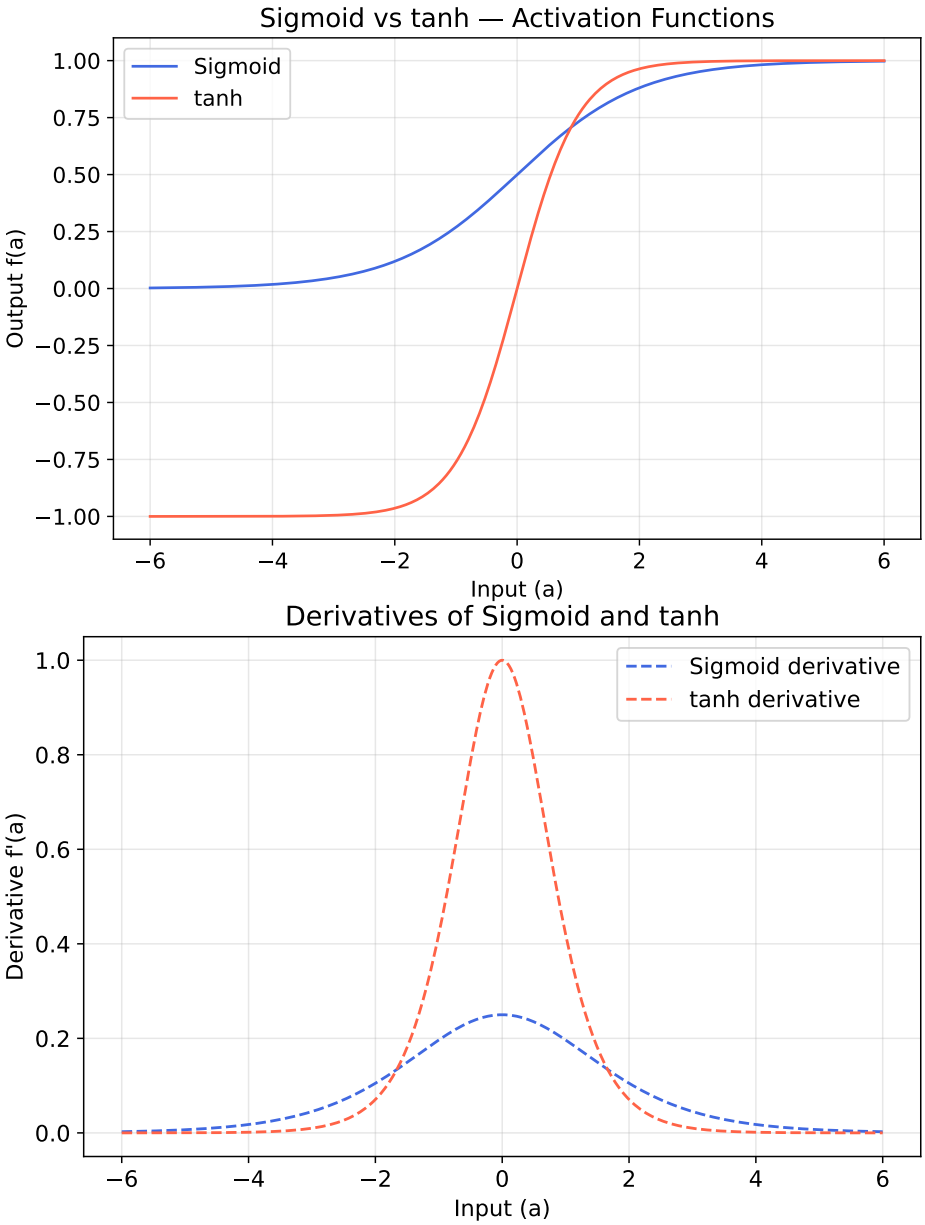


Figure 8: Hyperbolic Tangent (tanh) Activation Function and its Derivative.

The **zero-centered** output means activations can cancel each other out more easily, so the network doesn't get a constant "positive bias" in its gradient (a problem with sigmoid). This often leads to **faster convergence** during training.

When to use tanh

The tanh activation function is often preferred over the sigmoid function in hidden layers of neural networks, especially when the data is centered around zero. It is particularly useful in scenarios where:

- The **input** data is **normalized** to have a **mean of zero**.
- The **model requires faster convergence** during training.
- The **network is deep**, and the benefits of zero-centered activations help mitigate issues like vanishing gradients.

However, it's important to note that while tanh can be advantageous in many situations, it **still suffers from the vanishing gradient problem for very large or very small input values**. Therefore, in very deep networks, other activation functions like ReLU (Rectified Linear Unit) are often preferred.

2.3.3 Output Layer

The **output layer** is the *last* layer of the network, the one that produces the model's **final prediction**. Up to this point, the **hidden layers** have been learning to extract useful features (patterns, relationships, hierarchies). But the **output layer** translates all of that internal representation into the final, human-meaningful result. For example:

- A **continuous number** (e.g., house price) in **regression tasks** (e.g., predicting a numerical value).
- Or a **class label** (e.g., cat vs. dog) in **classification tasks** (e.g., categorizing images).

So, the **choice of activation function** in the output layer depends on the **type of output we want**. Exist several options:

- For **regression tasks**, where we want to predict a continuous value, we often use a **linear activation function** (or no activation function at all) in the output layer. This allows the network to produce a wide range of values.
- For **binary classification tasks**, where we want to classify inputs into two classes, we typically use the **sigmoid activation function** in the output layer. This squashes the output to a value between 0 and 1, which can be interpreted as a probability.
- For **multi-class classification tasks**, where we want to classify inputs into more than two classes, we often use the **softmax activation function** in the output layer. This produces a probability distribution over the classes, ensuring that the sum of the outputs equals 1.

The **design of the output layer** is crucial because it directly affects how well the network can perform its intended task. Choosing the appropriate activation function and structure for the output layer ensures that the network's predictions are meaningful and useful for the specific problem at hand.

2.3.3.1 Regression

In **regression problems**, we want the network to predict **a real-valued quantity**, something that can take *any* number, positive or negative. For example, predicting the price of a house based on its features (size, location, number of rooms, etc.) is a regression task. In this case, the **output layer** of the neural network typically consists of a **single neuron** that produces a **continuous output**, not categorical labels (we want a number, not a class like "expensive" or "cheap").

The output function

For regression tasks, we don't want to limit or distort the network's output. Therefore, the last layer simply uses a **linear activation** (page 57):

$$f(a) = a \quad \text{or equivalently} \quad y = w^T x + b$$

This means the output neuron just returns the raw weighted sum of its inputs, no squashing or thresholding.

If we used a **sigmoid** or **tanh** activation in the output layer, the output would be forced into $(0, 1)$ or $(-1, 1)$ ranges, respectively. This would be problematic for regression tasks where the target variable can take on a wide range of values. For example, if we're predicting house prices, we want the output to be able to represent any price, not just values between 0 and 1 (e.g., a house could cost $\$250,000$, which is far outside the range of a sigmoid output, or a temperature could be -10 degrees Celsius, which is outside the range of tanh). The **linear** activation allows any real number to be output, making it suitable for regression tasks.

Typical network setup for regression

A typical neural network for regression tasks has the following structure:

Component	Example
Hidden layers	Several, with nonlinear activations (e.g., ReLU, tanh).
Output layer	One neuron (for single output) with **linear activation**.
Loss function	**Mean Squared Error (MSE)** or **Mean Absolute Error (MAE)**.

Deepening: Mean Squared Error (MSE)

When our network predicts continuous values (like prices, temperatures, voltages, etc.), we need a way to measure **how far the predictions are from the real targets**. That's what a **loss function** does: it quantifies the prediction error. The **Mean Squared Error (MSE)** is the most

common one for regression tasks:

$$\text{MSE} = \frac{1}{N} \cdot \sum_{i=1}^{N} (y_i - t_i)^2 \tag{16}$$

Where:

- N is the number of data points (samples).
- y_i is the predicted value for the i-th sample.
- t_i is the true (target) value for the i-th sample.

The term $(y_i - t_i)^2$ is the **error** (difference between prediction and truth), and we **square** it to make all errors positive (avoid cancellation) and to penalize **larger mistakes more strongly** (e.g., an error of 10 counts 100 times more than an error of 1). Then we **average** over all samples to get the mean error per prediction.

So MSE measures how "spread out" our predictions are around the true values. A **lower MSE** means our model is doing a better job at predicting the continuous target variable.

Example 4: Example of MSE calculation

Suppose we have the following regression model:

Sample	t_i	y_i	$y_i - t_i$	Squared Error
1	2	3	+1	1
2	5	4	−1	1
3	6	8	+2	4
4	3	2	−1	1

To compute the MSE:

$$\text{MSE} = \frac{1}{4} \cdot (1 + 1 + 4 + 1) = \frac{1}{4} \cdot 7 = 1.75$$

So the Mean Squared Error for this model is 1.75, indicating the average squared difference between the predicted and true values.

About derivations, it is important to note that MSE is **differentiable**, which is crucial for training neural networks using gradient-based optimization methods (we will cover this in detail later). The derivative of MSE with respect to the predictions y_i is:

$$\frac{\partial \,\text{MSE}}{\partial \, y_i} = \frac{2}{N} \cdot (y_i - t_i) \tag{17}$$

So, the weight updates are proportional to how wrong each prediction is. It means, large errors produce larger gradients, leading to bigger

adjustments in the weights during training, which helps the model learn more effectively.

In summary, MSE tells us **how far off our predictions are on average**. It's like saying "*how wrong am I, squared and averaged?*" The squaring heavily punishes big mistakes, making MSE ideal when we care about precision in regression tasks.

Deepening: Mean Absolute Error (MAE)

The **Mean Absolute Error** measures the **average absolute distance** between predicted and true values:

$$\text{MAE} = \frac{1}{N} \cdot \sum_{i=1}^{N} |y_i - t_i| \tag{18}$$

Where:

- N is the number of data points (samples).
- y_i is the predicted value for the i-th sample.
- t_i is the true (target) value for the i-th sample.

Unlike MSE, which *squares* the difference, MAE simply takes the **absolute value** of the error. That means:

- Every error contributes proportionally to its magnitude (no squaring).
- Large errors don't explode quadratically, they contribute linearly.

So MAE measures the **average size of the mistakes**, regardless of direction.

Example 5: Example of MAE calculation

Using the same predictions as before (from Example on page 67), to compute the MAE:

$$\text{MAE} = \frac{1}{4} \cdot (1 + 1 + 2 + 1) = \frac{1}{4} \cdot 5 = 1.25$$

So the Mean Absolute Error for this model is 1.25, indicating the average absolute difference between the predicted and true values. Compared to MSE, MAE gives a more direct sense of the average error magnitude without squaring.

Regarding derivations, the MAE is **not differentiable** at points where the prediction equals the target (i.e., $y_i = t_i$) because of the absolute

value function. However, we can use the **subgradient** for optimization:

$$\frac{\partial |x|}{\partial x} = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{if } x < 0 \\ \text{undefined (but taken as 0)} & \text{if } x = 0 \end{cases} \tag{19}$$

So gradient updates from MAE are **constant in magnitude**. They don't depend on how far the prediction is from the truth. That's why MAE can converge slower but more robustly, especially in the presence of outliers (which can heavily skew MSE).

In summary, MAE tells us **how many units off my predictions are on average**, while MSE punishes larger errors more severely:

- **MSE** tries to **minimize variance**, forces the model to avoid large mistakes aggressively.
- **MAE** tries to **minimize average error**, focuses on overall robustness.

If our data has **outliers** (e.g., occasional very wrong samples), MAE is less distorted by them because it doesn't exaggerate their impact.

2.3.3.2 Binary Classification

In **binary classification**, the task is to decide **two possible outcomes**, for example *spam* vs *not spam* in email filtering, or *disease* vs *no disease* in medical diagnosis. The **output** of the network is typically a single neuron that produces a value between 0 and 1, representing the **probability** of one of the classes:

$$\mathcal{P}(y = 1 \mid x) \in [0, 1]$$

That is, "*how likely is this input to belong to class 1?*" The other class's probability can be derived as:

$$\mathcal{P}(y = 0 \mid x) = 1 - \mathcal{P}(y = 1 \mid x)$$

The output function

At the output layer, we typically have:

- **1 neuron**, because we only need one value (the probability of class 1).
- The **activation function** is usually the **sigmoid function** (or sometimes the **tanh function**), which maps any real-valued number into the range (0, 1), making it suitable for probability estimation.

The **sigmoid function** is defined as:

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

Or the **tanh function**:

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

Despite tanh outputting values in the range $(-1, 1)$, it can be scaled to (0, 1) for probability interpretation:

- $f(a) > 0 \Rightarrow$ class 1
- $f(a) < 0 \Rightarrow$ class 0

It's sometimes preferred due to its zero-centered output, which can help with optimization. However, the **sigmoid** function is **more commonly used** in binary classification tasks.

Typical network setup for binary classification

Component	Example
Hidden layers	Several, with nonlinear activations (e.g., ReLU, tanh).
Output layer	One neuron (for single output) with **sigmoid activation**.
Loss function	**Binary Cross-Entropy (log loss)**.

Deepening: Binary Cross-Entropy (BCE, Log Loss)

The goal in **binary classification** is to predict the *probability* that an input belongs to class 1, given by our network's sigmoid output:

$$\hat{y} = f(a) = \frac{1}{1 + e^{-a}} \quad \in (0, 1)$$

The true label t is:

- 1 if the sample belongs to class 1,
- 0 if it belongs to class 0.

The **Binary Cross-Entropy (BCE, Log Loss)** loss function measures the difference between the predicted probabilities $\hat{y}$ and the true labels t. It is defined as:

$$\text{BCE}\,(t, \hat{y}) = L = -\frac{1}{N} \cdot \sum_{i=1}^{N} \left[t_i \cdot \ln\left(\hat{y}_i\right) + (1 - t_i) \cdot \ln\left(1 - \hat{y}_i\right) \right] \tag{20}$$

Let's understand each term:

- N is the number of samples in the dataset.
- t_i is the true label for sample i (0 or 1, since it's binary classification).
- $\hat{y}_i$ is the predicted probability for sample i (output of the sigmoid).
- $\ln\left(\hat{y}_i\right)$ **penalizes** the model when it **predicts a low probability for the true class** (when $t_i = 1$).
- $\ln\left(1 - \hat{y}_i\right)$ **penalizes** the model when it **predicts a high probability for the false class** (when $t_i = 0$).
- If the true label is 1 ($t_i = 1$), the loss simplifies to $-\ln\left(\hat{y}_i\right)$. The model is **penalized** when it predicts a **small** probability for class 1.
- If the true label is 0 ($t_i = 0$), the loss simplifies to $-\ln\left(1 - \hat{y}_i\right)$. The model is **penalized** when it predicts a **large** probability for class 1.

So, **the closer the prediction is to the truth, the smaller the loss**.

Example 6: BCE Calculation Example

Let's consider a simple example with 4 samples:

Sample	True Label (t)	Predicted Probability ($\hat{y}$)
1	1	0.9
2	0	0.2
3	1	0.4
4	0	0.6

Now, we calculate the BCE loss for each sample:

- Sample 1: $[1 \cdot \ln(0.9) + (1-1) \cdot \ln(1-0.9)] = \ln(0.9) \approx -0.105$
- Sample 2: $[0 \cdot \ln(0.2) + (1-0) \cdot \ln(1-0.2)] = \ln(0.8) \approx -0.223$
- Sample 3: $[1 \cdot \ln(0.4) + (1-1) \cdot \ln(1-0.4)] = \ln(0.4) \approx -0.916$
- Sample 4: $[0 \cdot \ln(0.6) + (1-0) \cdot \ln(1-0.6)] = \ln(0.4) \approx -0.916$

Finally, we compute the average BCE loss over all samples:

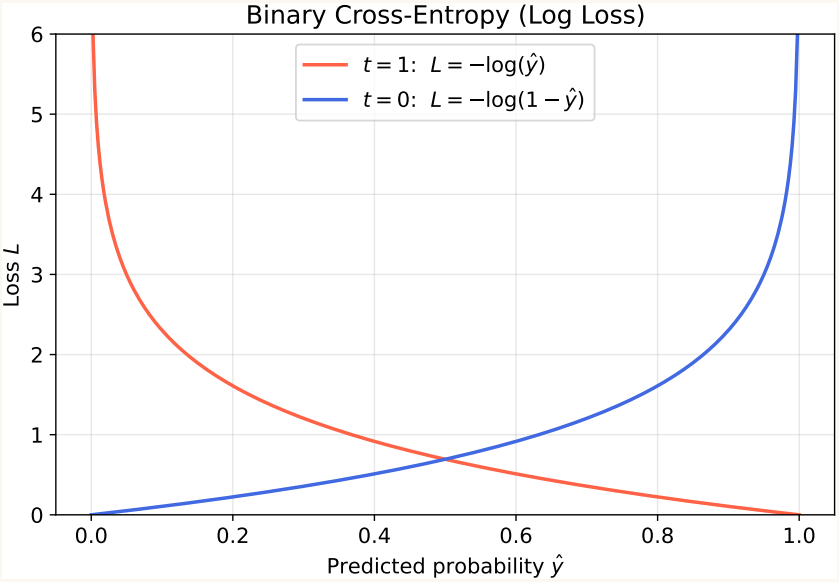
$$L = -\frac{1}{4}(-0.105 - 0.223 - 0.916 - 0.916) = -\frac{-2.16}{4} = 0.54$$

So, the BCE loss for this example is approximately **0.54**. This indicates that the model's predictions are not very accurate, as a lower loss value indicates better performance.

Cross-Entropy comes from **information theory**. It measures the **difference between two probability distributions**:

- The true distributions of the labels (0 or 1).
- The predicted distributions from the model (probabilities between 0 and 1).

Minimizing BCE is equivalent to **maximizing the likelihood** of our data under the model's predictions. So we're training the network to output probabilities that match the true labels as closely as possible.



The loss is **asymmetric**: wrong confident predictions get punished exponentially.

- Red curve ($t = 1$): Loss is low when predicted probability $\hat{y}$ is close to 1 (correct and confident), and high when $\hat{y}$ is close to 0 (incorrect).
- Blue curve ($t = 0$): Loss is low when predicted probability $\hat{y}$ is close to 0 (correct and confident), and high when $\hat{y}$ is close to 1 (incorrect).

Finally, BCE is **differentiable**, which is essential for training neural networks using gradient-based optimization methods. Its derivative with respect to a (the input to the sigmoid) is:

$$\frac{\partial L}{\partial a} = \hat{y} - t = f(a) - t \tag{21}$$

This derivative is used in backpropagation to update the network's weights during training.

In summary, **Binary Cross-Entropy** is the standard loss function for binary classification tasks in neural networks, effectively measuring the discrepancy between predicted probabilities and true binary labels, and guiding the training process to improve model performance. In simple words, it asks: "*how surprise would I be if the model's predicted probability were true?*" The less surprised (closer to 1 for correct class), the smaller the loss; the more surprised (model confident but wrong), the larger the penalty.

2.3.3.3 Multi-Class Classification

When we want the network to choose **one label among many**, for example to classify images of handwritten digits (0, 1, 2, ..., 9), we need the model to output a **vector of probabilities**, one for each possible class. For example, if the model is 70% sure that the image is a 3, 20% sure it is an 8, and 10% sure it is a 5, the output vector should be:

$$\hat{y} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0.7 \\ 0 \\ 0.1 \\ 0 \\ 0.2 \\ 0 \\ 0 \end{bmatrix}$$

Where each entry corresponds to the predicted probability of each class (from 0 to 9). To achieve this, exists two main techniques: **one-hot encoding** for the labels and the **softmax activation function** for the output layer.

How we represent targets: One-Hot Encoding

One-Hot Encoding is a simple way to represent **categorical variables** (things that take one of several discrete values, like *color*, *day of week*, or *class label*) in a numerical format that a neural network can understand.

The problem it solves. Neural networks work only with **numbers**, not words or symbols. So if our categories are, for example, *cat*, *dog*, and *bird*, we can't feed them directly into the network. We must convert each category into a **numeric vector**.

How it works. The naïve approach would be to assign each category a unique integer (e.g., *cat* = 0, *dog* = 1, *bird* = 2). But this is **misleading**, because the network would think that *bird* (2) is somehow "bigger" or "twice" a "dog" (1). That numerical relationship is meaningless and these categories have no inherent order. Instead, we create a **binary vector** for each class (category), where:

- The position corresponding to that class is set to 1 ("**hot**").
- All other positions are set to 0 ("**cold**").

So for our example with three categories, the one-hot encoded vectors would be:

- *cat* $\rightarrow [1, 0, 0]$
- *dog* $\rightarrow [0, 1, 0]$
- *bird* $\rightarrow [0, 0, 1]$

Each vector is called **one-hot vector** because exactly **one element is "hot"** (1) and all others are "cold" (0).

How we get probabilities: Softmax Activation Function

The **Softmax Activation Function** takes a vector of arbitrary real numbers (called *logits*) and turns it into a **probability distribution**, i.e. a vector of positive numbers that **sum to 1**:

$$\text{softmax}(a_i) = \frac{e^{a_i}}{\sum_{j=1}^{K} e^{a_j}} \tag{22}$$

Where:

- a_i is the i-th **element of the input vector** (logits).
- K is the total **number of classes**.
- e is the base of the natural logarithm (neperian constant).

Intuition. Each neuron in the output layer produces a **score**: a real number that can be positive, negative, or large. Softmax converts these scores into **relative probabilities** that express how confident the network is about each class:

- Large $a_i \rightarrow$ large $e^{a_i} \rightarrow$ **high** probability.
- Small $a_i \rightarrow$ small $e^{a_i} \rightarrow$ **low** probability.

The exponential function e^{a_i} magnifies differences between scores, so the biggest score gets *much more weight*, but every class still receives a small share.

Example 7

Suppose the output layer of a neural network produces the following logits for a 3-class classification problem:

$$a = \begin{bmatrix} 2.0 \\ 1.0 \\ 0.1 \end{bmatrix}$$

To convert these logits into probabilities using the softmax function, we first compute the exponentials:

$$e^a = \begin{bmatrix} e^{2.0} \\ e^{1.0} \\ e^{0.1} \end{bmatrix} \approx \begin{bmatrix} 7.389 \\ 2.718 \\ 1.105 \end{bmatrix}$$

Next, we sum these exponentials:

$$S = 7.389 + 2.718 + 1.105 \approx 11.212$$

Finally, we compute the softmax probabilities:

$$\text{softmax}(a) = \begin{bmatrix} \frac{7.389}{11.212} \\ \frac{2.718}{11.212} \\ \frac{1.105}{11.212} \end{bmatrix} \approx \begin{bmatrix} 0.659 \\ 0.242 \\ 0.099 \end{bmatrix}$$

Thus, the output probabilities for the three classes are approximately 65.9%, 24.2%, and 9.9%, respectively. The network is most confident that the input belongs to class 1.

Softmax acts like a "**competition**" between neurons:

- Each output neuron tries to **"win"** by having the highest score.
- The exponentials amplify the differences, making the highest score dominate.
- The normalization (dividing by the sum) ensures all probabilities add up to 1.

This is why it's called "softmax": it produces a **soft** version of the **maximum** function, where the highest score gets the most weight, but all classes still receive some probability (unlike a hard max which would assign 100% to the highest and 0% to all others).

🛠 Putting it all together

In a **K-class classification** problem, the network's final layer has:

- **K output neurons**, one for each class.
- Each neuron produces a **logit** (a_i) an unnormalized score.
- The **softmax function** converts these logits into a **probability distribution** over the K classes:

$$\hat{y}_i = \text{softmax}(a_i) = \frac{e^{a_i}}{\sum_{j=1}^{K} e^{a_j}}$$

So the network outputs a probability distribution over classes, all y_i are between 0 and 1, and they sum to 1. However, to train the network effectively, we also need a suitable loss function that works well with this setup (about training, we will discuss it later, but for now, let's focus on the loss function). This is where **Categorical Cross-Entropy (CCE)** comes into play.

The **Categorical Cross-Entropy (CCE)** loss function measures how close the predicted probability distribution $\hat{y}$ is to the true one-hot distribution $\mathbf{t}$:

$$\text{CCE}(\mathbf{t}, \hat{y}) = -\sum_{i=1}^{K} t_i \cdot \log(\hat{y}_i) \tag{23}$$

Because only the true class has $t_i = 1$ (all others are 0), this simplifies to:

$$\text{CCE}(\mathbf{t}, \hat{y}) = -\log(\hat{y}_c) \tag{24}$$

Where:

- c is the **index of the true class**.
- y_c is the **predicted probability for the true class**.

This means CCE penalizes the model when it assigns a low probability to the true class, encouraging it to predict higher probabilities for the correct class during training.

Why Softmax and CCE work well together? The combination of Softmax and CCE is powerful because:

- **Softmax produces a valid probability distribution**, which is exactly what CCE needs to compute the loss.
- **CCE** focuses the learning on **maximizing the probability of the true class**, which aligns perfectly with the goal of classification tasks.
- The **gradients** computed from CCE with respect to the logits are well-behaved, making training more stable and efficient. The drivative of CCE combined with Softmax is the following:

$$\frac{\partial\ (\text{CCE})}{\partial\, a_i} = \hat{y}_i - t_i \tag{25}$$

  This means the gradient is simply the difference between the **predicted probability** and the **true label**, which is **easy to compute** and interpret.

This synergy makes Softmax + CCE the **standard choice for multi-class classification problems in neural networks**. It is a generalization of the Sigmoid + BCE setup used for binary classification, extending the same principles to handle multiple classes effectively.

2.3.4 Neural Networks as Universal Approximators

In 1989, Kurt Hornik, Maxwell Stinchcombe, and Halbert White published a seminal paper titled "Multilayer Feedforward Networks are Universal Approximators" [2]. This groundbreaking work established that Feed-Forward Neural Networks (FNNs) with at least one hidden layer and non-linear activation functions can approximate any continuous function on compact subsets of $\mathbb{R}^n$ to any desired degree of accuracy, given sufficient neurons in the hidden layer:

> "*A single hidden layer feed-forward neural network with S-shaped activation functions can approximate any measurable function to any desired degree of accuracy on a compact set.*"

This theorem establishes the *theoretical power* of neural networks: given enough hidden neurons, an FNN can approximate **any continuous function** $f(x)$ over a bounded input domain. Let's define this more formally.

Theorem 1 (Universal Approximation Theorem)**.** *Let:*

$$f : \mathbb{R}^n \to \mathbb{R} \tag{26}$$

By any continuous function on a compact subset of $\mathbb{R}^n$ *(*$K \subset \mathbb{R}^n$*).*
Then, for any $\varepsilon > 0$*, there exists:*

- *A* ***single-hidden-layer neural network***
- *With* ***finite*** *number of neurons* J
- *And* ***non-linear activation*** $\sigma(\cdot)$ *(e.g., sigmoid, tanh, ReLU, etc.)*

Such that for all $x \in K$*:*

$$\left| f(x) - \sum_{j=1}^{J} w_j^{(2)} \cdot \sigma \cdot \sum_{i=1}^{n} \left(w_{ji}^{(1)} x_i + b_j \right) \right| < \varepsilon \tag{27}$$

Where:

- $w_{ji}^{(1)}$ *are the* ***weights*** *from* ***input layer to hidden layer****.*
- b_j *are the* ***biases*** *of the* ***hidden layer*** *neurons.*
- $\sum_{i=1}^{n} \left(w_{ji}^{(1)} x_i + b_j \right)$ *is the* ***input to the hidden layer*** *neuron* j*.*
- $\sigma(\cdot)$ *is the* ***non-linear activation function*** *applied at* ***hidden*** *layer neurons.*
- $w_j^{(2)}$ *are the* ***weights from hidden layer to output layer****.*
- *The output is the result of the neural network for input* x*.*

In simpler terms, any continuous can be represented by a neural network with just **one hidden layer**, if that layer has enough neurons and uses a non-linear activation function.

Why it works (intuition)

Each hidden neuron with non-linear activation acts like a **basis function** (similar to how sine and cosine functions can approximate any waveform in Fourier series). By combining enough of these basis functions (hidden neurons), the neural network can approximate complex functions by adjusting the weights and biases.

Imagine we have an unknown function $f(x) = \sin(x)$. And we want our neural network to **learn** this function. So, in other words, we want our neural network to approximate $f(x)$ as closely as possible ($\hat{f}(x) \approx f(x)$). Let's take a **single neuron** with a non-linear activation function (e.g., sigmoid):

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

If we plot this, it looks like an S-shaped curve (page 59):

- Almost 0 for large negative inputs.
- Almost 1 for large positive inputs.
- Smoothly transitions between 0 and 1 around input 0.

When we apply this neuron to a **linear combination of** x:

$$\sigma(w \cdot x + b)$$

We get a *shifted and stretched S-curve* along the x-axis. Now imagine we have **many hidden neurons**, each with their own weights and biases:

$$\hat{f}(x) = \sum_{j=1}^{J} w_j^{(2)} \cdot \sigma(w_j^{(1)} \cdot x + b_j)$$

Each neuron produces its own "bump" or "S-step" at a different location. When we **add them together**, those bumps **stack up and blend**, creating any curve shape we want. And that's the whole trick! Just like adding sinusoids can approximate any periodic signal (Fourier series), adding non-linear S-shaped functions can approximate any continuous curve. Note that non-linearity is crucial. If we used only linear activations and stacked them, the result would still be a linear function. This would collapse the network's expressive power.

Important note

The Universal Approximation Theorem guarantees that a neural network **can** approximate any continuous function, but it does not tell us **how to find** the right weights and biases to do so. In practice, training a neural network to approximate a specific function requires effective optimization algorithms (like gradient descent) and sufficient training data. Additionally, while a single hidden layer is theoretically sufficient, deeper networks (with more hidden layers) often learn more efficiently and generalize better in practice.

2.4 Learning and Optimization

Let's retrace what we've built so far step by step:

1. We started with the **historical context**, understanding why we want machines to "learn" like brains, transitioning from symbolic AI to data-driven learning.
2. Next, we explored the **Perceptron**, the simplest computational neuron, which introduced us to linear decision boundaries and Hebbian learning.
3. However, we also learned about the **limitations of the Perceptron**, particularly its inability to solve non-linear problems like XOR.
4. To overcome these limitations, we delved into **Feed-Forward Neural Networks (FNNs)**, discovering how multi-layer networks with hidden layers and nonlinear activations can model complex functions.
5. Finally, we touched on the **Universal Approximation Theorem**, which assures us that even a single hidden layer is theoretically sufficient to approximate any function (though in practice, deeper networks often perform better).

Now we know **what** the architecture can represent. But we haven't yet learned **how** to *find the right weights* that make it represent what we want. And that's *exactly* why this section begins.

After defining the structure of a neural network, we must **teach it** to perform a task, such as classifying images or predicting values. This teaching process is called **learning** or **training** (or simply *learning by optimization*). Think of the journey like this:

$$\text{Architecture} \rightarrow \text{Function Space} \rightarrow \text{Optimization} \rightarrow \text{Learning}$$

We've defined the **function space** (what kinds of functions the network can represent), and now, we must **search inside that space** for the specific function that matches our data. This search is done through **optimization algorithms** that adjust the network's weights based on the data we provide. So, in this section, we'll answer three big questions:

1. ***How does a neural networks learn?*** (page 81) By comparing predictions with known targets (supervised learning).
2. ***How do we measure "how wrong" it is?*** (page 84) Through *loss functions* (some of which we have already encountered in the output layer design).
3. ***How do we improve it?*** (page 88) Through *optimization algorithms* like gradient descent and (later) backpropagation.

2.4.1 Supervised Learning and Training Dataset

This section introduces the **formal setup** of *Supervised Learning* in the context of neural networks. It defines **what data to use**, **what we want the network to learn**, and **how we measure learning success**. It's the *conceptual skeleton* that the later mathematical tools (loss, gradient descent, backpropagation) will stand on.

What is Supervised Learning?

Supervised Learning is a **machine learning paradigm** where the algorithm learns **from examples that include both the input and the correct output**. In other words, it learns **under supervision** from labeled data.

The basic idea is simple. We give the model a set of **training examples**:

$$\mathcal{D} = \{(x_1, t_1), (x_2, t_2), \ldots, (x_N, t_N)\}$$

Formally, a **dataset** is:

$$\mathcal{D} = \left\{ (x_i, t_i) \right\}_{i=1}^{N} \tag{28}$$

Where:

- x_i is the **input data** (e.g., an image, temperature readings, pixels, sensor values, etc.).
- t_i is the **target output** (the *label* or *ground truth* we want the model to predict).
- N is the total number of training examples.

The model (in our case, a neural network) tries to learn a **function** $f(\cdot)$ such that:

$$f(x_i) \approx t_i \quad \text{for all } i = 1, 2, \ldots, N$$

For all examples in the training set. In other words, to find a function $f(x)$ that not only fits the training data but also **generalizes** well to unseen data (i.e., it can correctly predict outputs for new inputs not in the training set). Formally, a **model** is a function:

$$g(x; w) \text{ with parameters } w \tag{29}$$

Where:

- $g(\cdot; w)$ is the model (neural network) with parameters w (weights and biases).
- The goal is to find the optimal parameters w^* such that:

$$w^* = \arg\min_w E(w) \tag{30}$$

  Where $E(w)$ is a **loss function** that measures how far predictions $g(x_i; w)$ are from the true targets t_i across the training set.

The method is called **supervised** because the learning process is **guided**: each input x comes with the **correct answer** t. The network uses it to know whether it was right or wrong, and to adjust its weights accordingly.

Independently by the paradigm used (supervised, unsupervised, reinforcement learning), with **Training** we mean the **process of adjusting weights w so that the network reproduces the mapping between inputs and outputs seen in the data**. This is done by **minimizing a loss function** that quantifies the difference between the network's predictions and the true targets in the training dataset.

Neural Networks as a Parametric Model

In general, a neural network can be written as a **parametric function**:

$$y\left(x;w\right)=g\left(x,w\right) \tag{31}$$

Where:

- x is the input features (data).
- w is the set of parameters (weights and biases in all layers) of the network.
- $y\left(x;w\right)$ is the output of the network (the prediction for input x given parameters w).
- ; indicates that y depends on both x and w.
- $g(\cdot,\cdot)$ represents the entire computation performed by the neural network (all layers, activations, etc.). See above equation 29.

We want $y\left(x_n;w\right)$ to be as close as possible to the target t_n for each training example (x_n,t_n) in the dataset $\mathcal{D}$. This is achieved by **optimizing the parameters** w to minimize a **loss function** that measures the discrepancy between predictions and targets across all training examples. Formally, find parameters w^* that minimize a loss function $E(w)$:

$$w^*=\arg\min_w E\left(w\right) \tag{32}$$

Where $E(w)$ quantifies the error between $y\left(x_n;w\right)$ and t_n for all training examples. The loss function measures how wrong the model is across all training examples:

$$E\left(w\right)=\sum_{n=1}^{N}\ell\left(t_n,y\left(x_n;w\right)\right) \tag{33}$$

Where $\ell(\cdot,\cdot)$ is a loss function that quantifies the error for a single example (e.g., Mean Squared Error for regression, Cross-Entropy Loss for classification).

The power of this setup

This framework allows us to covers a wide range of tasks:

- **Regression**: Predicting continuous values (e.g., house prices, temperature).
- **Classification**: Assigning inputs to discrete categories (e.g., spam vs. not spam, image recognition).
- **Function Approximation**: Learning complex mappings from inputs to outputs.

By defining the problem in terms of inputs, targets, and a loss function, we can apply various optimization algorithms (like gradient descent) to train the neural network effectively. So, we must care only about:

- The **structure of the network** (layers, activation functions).
- The **choice of loss function** (depends on the task).
- The **optimization algorithm** to minimize the loss.

This abstraction makes neural networks a versatile tool for many machine learning problems.

Example 8: Analogy

Imagine a student (the network) learning to solve math exercises.

- **Inputs** x_n are the exercises given to the student by the teacher.
- **Targets** t_n are the correct answers provided by the solution book.
- **Network** $g(x, w)$ is the student's method of solving the exercises, which depends on their current knowledge (parameters w).
- **Loss function** $E(w)$ measures how many answers the student got wrong compared to the solution book.
- **Optimization** (training) is the process of the student studying and adjusting their methods (updating w) to minimize mistakes on future exercises.

Over time, with enough practice (training examples), the student improves their ability to solve new exercises correctly (generalization). So training means making fewer mistakes over time by adjusting reasoning (weights).

2.4.2 Error Minimization and Loss Function (SSE)

To teach a neural network, we need a way to **measure how wrong** it is. That measure is the **error (loss) function**. Once defined. we can **minimize** it by adjusting the weights, and the process **is** the essence of learning.

Definition 6: Error Function

Given a **training set**:

$$\mathcal{D} = \left\{ (x_n, t_n) \right\}_{n=1}^{N} \tag{34}$$

And the network's predictions:

$$y_n = g\left(x_n; w\right) \tag{35}$$

The **Error Function** $E(w)$ measures the total discrepancy between all predictions and their true targets:

$$E(w) = \sum_{n=1}^{N} \text{Loss}(t_n, y_n) \tag{36}$$

Where $\text{Loss}\left(\cdot\right)$ is the per-sample difference between the predicted and actual value.

Definition 7: Loss Function

A **Loss Function** (sometimes called **error** or **Cost Function**) is a **mathematical function that quantifies how wrong a model's predictions are** compared to the true (target) values. In other words:

$$\text{Loss}(t, y) = \text{scalar measure of discrepancy between } t \text{ and } y \tag{37}$$

Where t is the true target value, and y is the model's prediction. The loss function **outputs a single number representing how bad the prediction is**; lower values indicate better predictions. During training/learning, the network tries to **minimize** this loss by adjusting its weights, to reduce its mistakes.

It is strictly related to the **Error Function** $E(w)$, which aggregates the loss over the entire training set to give a total measure of how well the model is performing. Indeed, for a single training example (x_n, t_n), we have:

$$L_n = \text{Loss}\left(t_n, g\left(x_n; w\right)\right) \tag{38}$$

Where L_n is the loss for sample n, t_n is the true target, and $g\left(x_n; w\right)$ is the model's prediction for input x_n with weights w. And the total error

function over all N samples is:

$$E(w) = \sum_{n=1}^{N} L_n = \sum_{n=1}^{N} \text{Loss}\left(t_n, g\left(x_n; w\right)\right)$$

So, the **loss function measures the error for one sample**, while the **Error Function sums these losses over the entire dataset** to give a total error measure.

The simplest and most classic choice: SSE

Exists many choices for the loss function. The simplest and most classic is the **Sum of Squared Errors (SSE)**. In early neural networks (and still often in regression problems), the **Sum of Squared Errors (SSE)** was the standard loss function:

$$E(w) = \sum_{n=1}^{N} \text{Loss}(t_n, y_n) = \sum_{n=1}^{N} \left[t_n - g\left(x_n; w\right)\right]^2 \tag{39}$$

- t_n is the true target for sample n.
- $g\left(x_n; w\right)$ is the network's prediction for input x_n.

Similar to the Mean Squared Error (MSE, page 67), the SSE squares the differences to makes all errors **positive** (so under- and over-predictions both count) and **emphasizes large errors** (penalizes them more heavily). Also, squaring makes the error function **differentiable**, which is crucial for optimization algorithms like gradient descent.

Minimizing the SSE means finding the weights w that make the network's predictions as close as possible to the true targets across the entire training set. Formally, learning becomes finding the set of weights w^* that minimize the total error:

$$w^* = \arg\min_{w} E(w)$$

Each weight w_i acts like a small knob controlling part of the network's behavior. We tweak these knobs slightly so that the network's predictions move closer to the true outputs. When all knobs are adjusted such that $E(w)$ is as small as possible, the network has **learned** the function.

Geometric Interpretation

Minimizing the error means **finding a point in parameter space w where the error surface $E(w)$ reaches its minimum**. We can visualize $E(w)$ as a **landscape**: valleys represent low error (good predictions), and hills represent high error (bad predictions). The learning process is like navigating this landscape to find the lowest valley, which corresponds to the optimal weights w^* that minimize the error.

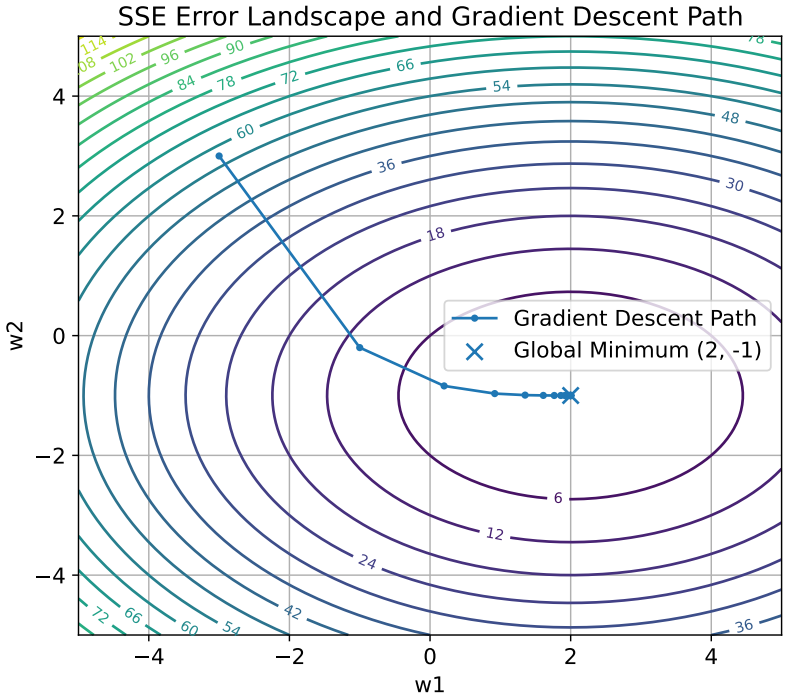


Figure 9: How the **Sum of Squared Errors (SSE)** behaves as a function of the **model parameters (weights)**, and how **gradient descent** moves step by step toward the minimum error point. The x-axis w_1 and y-axis w_2 represent two weights (parameters) of the model/ Each contour line shows **all combinations of weights** (w_1, w_2) that produce the **same total error** $E(w_1, w_2)$. The loss function we use:

$$E\left(w_1, w_2\right) = \left(w_1 - 2\right)^2 + 2\left(w_2 + 1\right)^2$$

Those small points connected by a line represent the **path that gradient descent follows** over time. Starting from an initial guess, each step moves downhill toward lower error, reaching the minimum point where the error is lowest (point $(2, -1)$ in this case). In a real network, the number of weights isn't just two but can be thousands or millions, making the error surface a high-dimensional landscape. We can't visualize that directly, but this contour map is an **analogy** to help understand how optimization algorithms like gradient descent navigate the error surface to find the best weights.

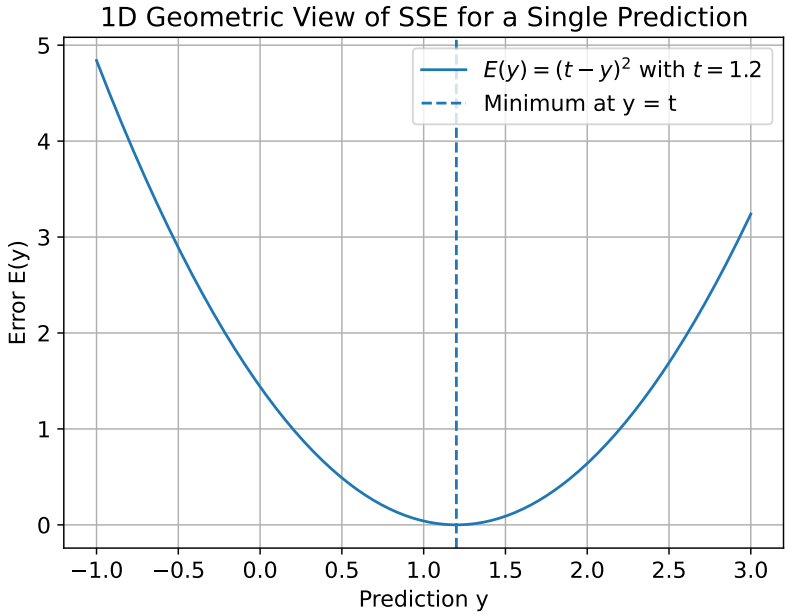


Figure 10: The **error function** for **a single training example** using the **Sum of Squared Errors (SSE)** loss: $E(y) = (t - y)^2$, where t is the true target and y is the model's prediction. The x-axis represents the model's prediction y (all possible output values the model could predict for this input sample), and the y-axis shows the corresponding error $E(y)$ (how wrong the model would be for each possible prediction). This graph shows **how the error changes** as the model output moves away from the correct answer. It is a **parabola**, because the error grows quadratically as we move away from the correct value. At $y = t$, the error is zero (the model predicts perfectly) because $E(t) = (t - t)^2 = 0$. As y moves away from t, the difference grows, and squaring makes it **positive and larger**. The dashed vertical line marks the **minimum point** where the prediction perfectly matches the target ($y = t$, zero error).

Relation to Statistical Foundations

The squared error has a nice **probabilistic interpretation**. If we assume that the target values t_n are generated by a model with **Gaussian noise**:

$$t_n = g(x_n; w) + \epsilon_n, \quad \epsilon_n \sim \mathcal{N}(0, \sigma^2)$$

Then minimizing the sum of squared errors (SSE) is equivalent to **Maximum Likelihood Estimation (MLE)** of the weights w. In other words, by minimizing SSE, we are finding the weights that make the observed data most probable under the assumed Gaussian noise model. So the SSE is not arbitrary choice; it has a solid statistical basis when the noise in the data is Gaussian.

2.4.3 Gradient Descent Basics

We've defined the learning goal:

$$w^* = \arg\min_w E(w)$$

Where $E(w)$ is our **error (or loss) function**, for instance:

$$E(w) = \sum_{i=1}^{N} (t_n - g(x_n; w))^2$$

The idea of **gradient descent** is to find this minimum *iteratively*, by moving the parameters w step by step in the direction that **reduces** the error.

The Concept of the Gradient and the Key Idea of Gradient Descent

Let's start simple. For a function of one variable $E(w)$, the **derivative** $\frac{\partial E}{\partial w}$ at a point w tells us the slope of the function:

- If positive $\rightarrow$ $E(w)$ increases as w increases.
- If negative $\rightarrow$ $E(w)$ decreases as w increases.
- If zero $\rightarrow$ $E(w)$ is flat (local minimum or maximum).

In higher dimensions (e.g., multiple weights $w_1, w_2, \ldots, w_d$), we generalize the derivative to a **vector** of partial derivatives:

$$\nabla E(w) = \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \vdots \\ \frac{\partial E}{\partial w_d} \end{bmatrix}$$

This vector, the **gradient** of E at w, points in the direction of **steepest ascent** of the function (the **direction** in which $E(w)$ **increases the most rapidly**).

So, if we want to **minimize** $E(w)$, we should move in the direction of **steepest descent**, which is the **opposite direction** of the gradient, i.e., $-\nabla E(w)$. That's why it's called **gradient descent**!

Definition 8: Gradient Descent

Gradient Descent is an **iterative optimization algorithm** used to find the set of parameters (weights) that **minimize a loss function**. In simple words, it is the process by which a neural network *learns* by **repeatedly adjusting its weights in the direction that reduces**

the loss the most.

Formally, let w be the vector of weights, and $E(w)$ be the loss function. The **gradient** with respect to the weights is given by $\nabla E(w)$:

$$\nabla E(w) = \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \vdots \\ \frac{\partial E}{\partial w_k} \end{bmatrix} \tag{40}$$

This vector points in the **direction of steepest increase** of $E(w)$. To *minimize* the loss, we go in the **opposite direction** of the gradient. That gives the update rule for the weights, known as **Core Learning Rule**:

$$w_{k+1} = w_k - \eta \nabla E(w_k) \tag{41}$$

Where:

- w_k is the weight vector at iteration k.
- $\nabla E(w_k)$ is the gradient (slope) of the error function at w_k.
- η is the **learning rate** (step size), a small positive scalar that controls the step size.
- w_{k+1} is the updated weight vector after taking a step in the direction of steepest descent.

It is called Core Learning Rule because it is the **fundamental principle underlying how neural networks learn from data by adjusting their weights to minimize error**. Everything that comes next (like backpropagation) are all *variants or extensions* of this exact rule. They all keep this same pattern:

new param = old param−(some learning rate)×(some form of gradient)

The only difference is *how* the gradient or learning rate is computed or adjusted. It is a sort of **DNA of learning** in neural networks.

The Neural Network Case

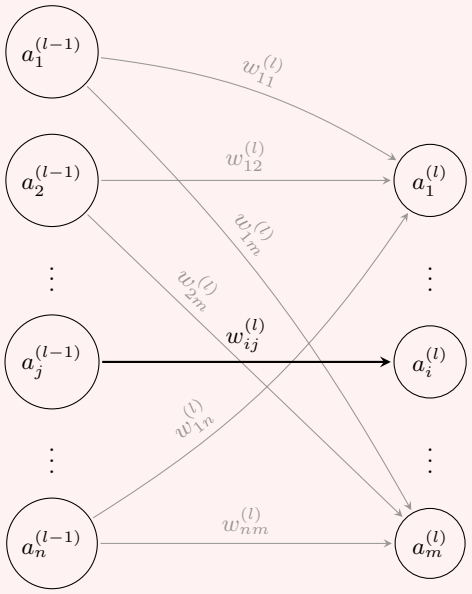
The previous definition is **general**, and it doesn't care whether w is one number, a list or a tensor of weights inside a neural network. However, sice we are in the context of neural networks, let's introduce some notation specific to them.

In a neural network, w isn't just one parameter vector (one weight vector), but rather a **collection of all weights and biases across all layers**:

$$w = \left\{ w_{ij}^{(l+1)}, w_{ij}^{(l+2)}, \ldots, w_{ij}^{(L)}, b_i^{(l+1)}, b_i^{(l+2)}, \ldots, b_i^{(L)} \right\} \tag{42}$$

Where:

- L is the **total number of layers in the network**.
- Each $w_{ij}^{(l)}$ is the weight connecting neuron j in layer $l-1$ to neuron i in layer l:



- Each $b_i^{(l)}$ is the bias for neuron i in layer l.

The **gradient** $\nabla E(w)$ then contains the partial derivatives of the loss function with respect to **each individual weight and bias** in the network:

$$\nabla E(w) = \left\{ \frac{\partial E}{\partial w_{ij}^{(l)}}, \frac{\partial E}{\partial b_i^{(l)}} \right\} \tag{43}$$

The **core learning rule** still applies, but now we update **each weight and bias** individually:

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \frac{\partial E}{\partial w_{ij}^{(l)}} \quad \text{and} \quad b_i^{(l)} \leftarrow b_i^{(l)} - \eta \frac{\partial E}{\partial b_i^{(l)}} \tag{44}$$

This is how **gradient descent** is applied in the context of training neural networks.

How it works (intuitively)

Intuitively, gradient descent works as follows:

1. Start from an **initial guess** for the weights w_0 (often random).
2. Compute the **gradient** $\nabla E(w_0)$ of the loss function at the current weights:

$$\nabla E(w_0) = \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \vdots \\ \frac{\partial E}{\partial w_d} \end{bmatrix}_{w=w_0}$$

3. Do a step **against** that gradient to update the weights using the **core learning rule**:

$$w_{k+1} = w_k - \eta \nabla E(w_k) \quad \Rightarrow \quad w_1 = w_0 - \eta \nabla E(w_0)$$

   Then, the new weights w_1 should yield a **lower loss** $E(w_1) < E(w_0)$. If we were on a neural network, we would update **all weights and biases** similarly:

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \frac{\partial E}{\partial w_{ij}^{(l)}} \quad , \quad b_i^{(l)} \leftarrow b_i^{(l)} - \eta \frac{\partial E}{\partial b_i^{(l)}}$$

4. Repeat until: the gradient becomes small (close to zero), or we reach a maximum number of iterations.

Convergence Speed and False Positives. The speed at which gradient descent converges to the minimum depends on:

- The **shape** of the loss function (e.g., steepness, curvature). It strongly affects how gradient descent behaves:
  - **Convex surface** (like a bowl): Gradient descent always converges there (since there is a **single global minimum**). However, if η is too large, it may oscillate around the minimum.
  - **Non-convex surface** (like many hills and valleys): There may be **multiple local minima** and **saddle points**. Gradient descent might:
    - Get stuck in a **local minimum** (the derivative is zero, but it is not the global optimum).
    - Oscillate in flat regions.
    - Move very slowly along narrow valleys.
- The **learning rate** η.

  The **learning rate** η is crucial: it controls **how big a step** we take each iteration.

  - η too **small** → **Learning is very slow** because we take tiny steps (many iterations needed).
  - η too **large** → **May overshoot the minimum** and even diverge (loss increases instead of decreasing).
  - Choosing a good η is often done via **experimentation** or using techniques. However, if η is chosen well, gradient descent can efficiently find a good set of weights that minimize the loss function.

  Don't worry if this seems abstract now; we'll later learn that **adaptive optimizers** (like ***momentum***, ***Adam***, etc.) are more robust ways to handle this.

- The **initial weights** w_0. In **convex** functions (like a parabola), there is a global minimum, and *gradient descent* will converge to it. In **non-convex** functions (like many neural network loss landscapes), there may be multiple local minima, and gradient descent may converge to one of them depending on the starting point. So in practice, we often run gradient descent multiple times with different initial weights to find a good solution.

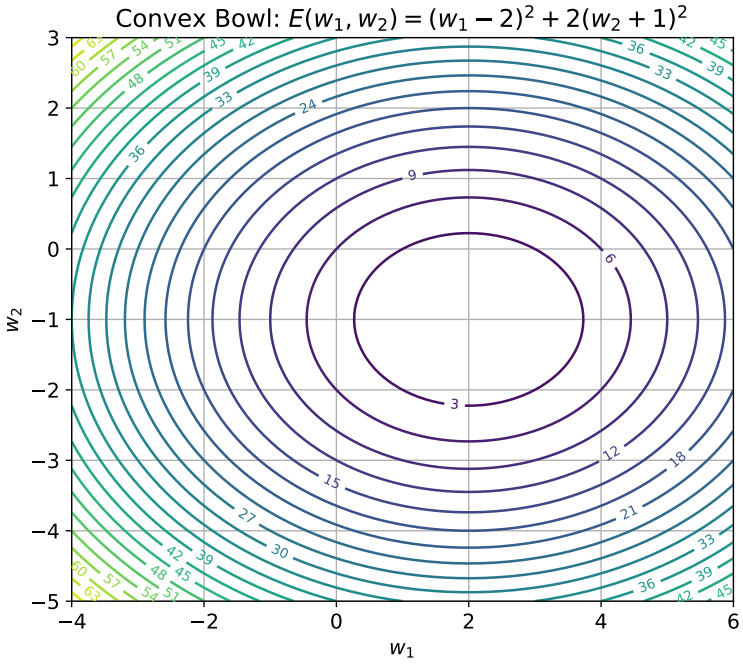


Figure 11: Example of a **convex quadratic surface** (bowl-shaped). The contour lines (ellipses) represent levels of constant error $E(w_1, w_2)$. There is a **single global minimum** at the center of the bowl ($w_1^* = 2, w_2^* = -1$), and outer ellipses represent higher error values. This is the ideal scenario for gradient descent, as it will always converge to the global minimum regardless of the starting point (similar to Figure 9, page 86).

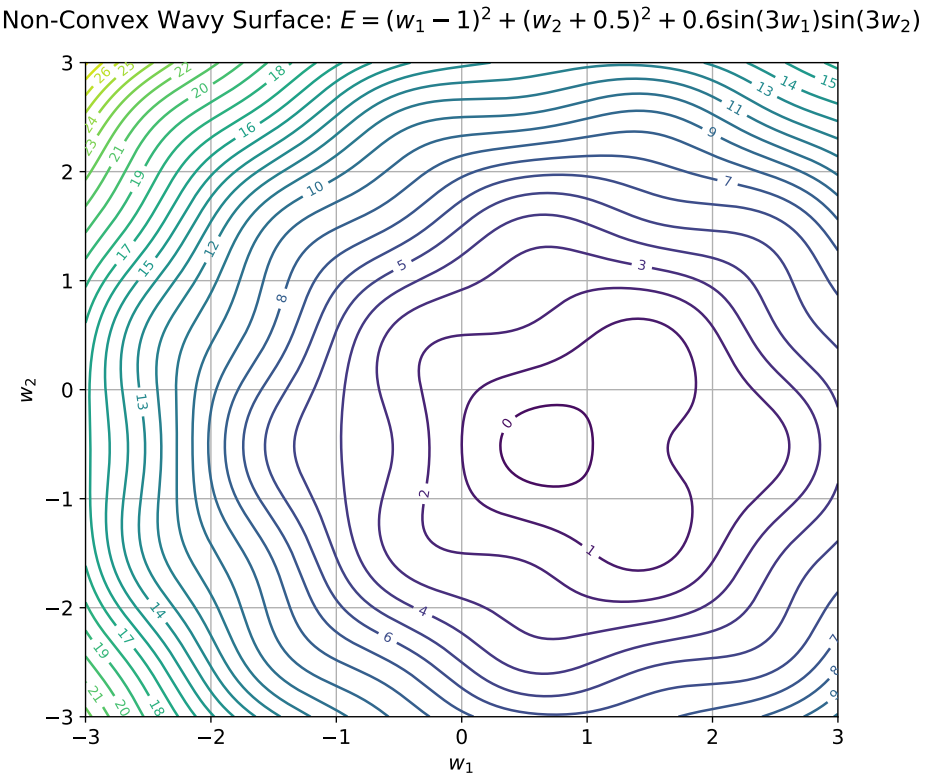


Figure 12: Example of a **non-convex surface** with multiple local minima and saddle points. The loss function $E(w_1, w_2)$ has two components: (1) the **quadratic bowl** that pushes weights toward the global minimum, and (2) the **sinusoidal term** that introduces *oscillations* in the surface. Those oscillations create **small waves** (bumps and dips). Each dip (a small valley) is a *local minimum*, and each bump is a *local maximum* or ridge. This is what happens in **non-linear models** like deep neural networks, where the composition of many layers and activations makes the error surface very complex (and non-convex). There are thousands or millions of such local minima, making optimization challenging. Gradient descent may get trapped in one of these local minima instead of finding the global minimum. However, in practice, many local minima have **similar performance**, so this isn't always bad.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[2] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[3] Matteucci Matteo. Artificial neural networks and deep learning. Slides from the HPC-E master's degree course on Politecnico di Milano, 2025-2026.

Index