Foundations and Applications of Neural Networks

Latreche Sara

Contents

1 Course Goals

- Understand the theory behind neural networks.
- Implement neural networks from scratch in Python.
- Apply networks to real-world datasets and challenges.
- Explore architectures like CNNs, RNNs, and Transformers.

2 The Breakthrough: A Dream to Mimic the Mind

Figure 1: *
Ada Lovelace (1815–1852) – a mind centuries ahead of her time

Note

From imagination to implementation—this is the story of how humans began their quest to build thinking machines.

Before there were transistors, neural networks, or even the word "computer" in its modern sense, there was **Ada Lovelace**—a mathematician, writer, and visionary.

Decades before the first modern computers were built, she imagined something astonishing: that machines might one day do more than calculate—they might *create*.

While translating an article on Charles Babbage's proposed Analytical Engine, she added a series of notes—longer than the article itself. In one of these, she made a radical suggestion:

> *"Numerous fundamental relations of music can be expressed by those of the abstract science of operations, such that a machine could compose elaborate and scientific pieces of music of any degree of complexity or extent."*
>
> — Ada Lovelace, 1843

This was not a dream of automation—it was a dream of intelligence. Lovelace believed that

the Analytical Engine, though designed to manipulate numbers, could one day manipulate symbols and ideas.

Note

Her insight wasn't just technical—it was philosophical. She foresaw machines that could be not only precise, but also creative. In essence, she predicted the core vision of artificial intelligence.

Reflection

Ada Lovelace wasn't merely the first programmer—she was the first to dream of machines that could one day mimic the complexity of human thought. She planted the seed for a future where computers wouldn't just follow instructions—they might compose symphonies.

Alan Turing: The Father of Theoretical AI

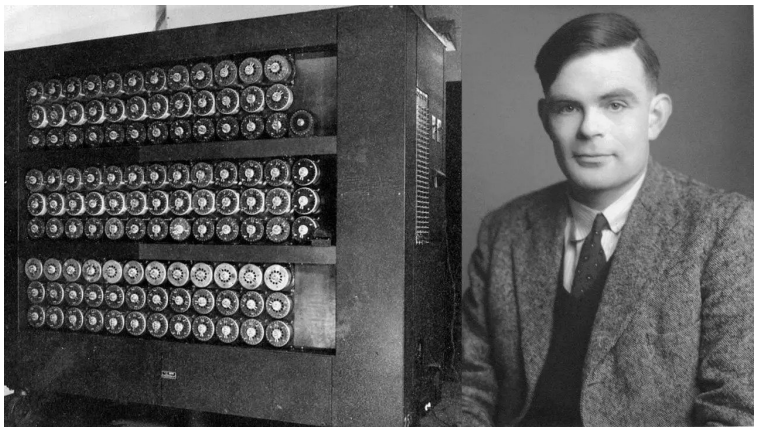


Figure 2: *
Alan Turing (1912–1954) – Architect of modern computing

Note

Turing didn't just imagine thinking machines—he laid the theoretical and philosophical groundwork for everything AI has become.

Theoretical Work and the Universal Machine

In 1935, long before digital computers existed, **Alan Turing** described an abstract computing device with infinite memory and a scanner that could read, write, and modify symbols based on rules—this was the birth of the ***universal Turing machine***.

Key Insight

Turing's machine was more than a thought experiment—it introduced the concept of a stored-program computer. His model suggested that a machine could, in theory, read and modify its own instructions. Today, every modern computer is a descendant of this idea.

Wartime Genius and the Seeds of AI

During World War II, Turing worked as a cryptanalyst at Bletchley Park, helping crack the German Enigma code using electromechanical machines. But even amidst this critical work, he contemplated deeper questions: *Could machines learn from experience? Could they solve problems by adapting over time?*

His colleague Donald Michie later recalled how Turing often discussed the idea of heuristic learning—a core idea in today's AI. In 1947, Turing publicly declared:

> *"What we want is a machine that can learn from experience. The possibility of letting the machine alter its own instructions provides the mechanism for this."*

These ideas appeared more formally in his 1948 report, *"Intelligent Machinery"*, which introduced neural-like networks and the idea of machines capable of self-improvement. Though unpublished at the time, it laid the foundations for later developments in connectionist AI.

Chess and the Mind of the Machine

Turing viewed chess as a proving ground for artificial intelligence. He believed it required planning, memory, and adaptation—traits that a thinking machine would need to replicate.

Example

Turing wrote one of the first chess algorithms without having a computer to run it. He simulated the machine by hand, calculating its moves manually!

In 1945, Turing predicted that machines would one day play "very good chess." He was right: in 1997, IBM's **Deep Blue** defeated world champion **Garry Kasparov**, making headlines around the world.

Reflection

Although chess engines like Deep Blue reached superhuman performance, their success came more from engineering brute-force search than from simulating human thought. As Noam Chomsky quipped, a bulldozer beating a human in weightlifting isn't proof of intelligence—it's a different kind of power.

The Turing Test: A Practical Definition of Intelligence

In 1950, Turing sidestepped the endless debate about defining intelligence and proposed a simple, elegant experiment—**The Imitation Game**, now known as the **Turing Test**.

What Is the Turing Test?

The test involves three players: a computer, a human, and a human judge. The judge converses with both (via text) and must decide which is the machine. If the computer can reliably fool the judge, it is said to have passed the test.

Turing imagined that by the year 2000, machines might be able to fool 30% of judges. For decades, no program came close.

Then, in 1991, the **Loebner Prize** was introduced—an annual competition awarding $2000 for the most humanlike AI in a Turing-style test. No one ever won the grand prize of $100000, as no machine truly passed the undiluted version of the test.

GPT-4 and the Modern Turning Point

In late 2022 and throughout 2023, the release of **ChatGPT** and **GPT-4** reignited debate about machine intelligence. Some, like Max Woolf, claimed GPT had "passed the Turing test," citing its uncanny humanlike fluency. But experts argued that since GPT-4 often reveals it is an AI, or lacks persistent memory and understanding, it hadn't truly passed the original challenge.

Note

Turing once dreamed of machines that could learn, adapt, and communicate. While GPT-4 may not "think" like us, it certainly imitates us better than any system before.

Reflection

Alan Turing laid the intellectual foundations of AI before the hardware even existed. His ideas on learning, adaptation, intelligence, and even self-modifying machines are not relics—they are living parts of today's neural networks and large language models.

3 From Biology to Circuits: Mimicking the Brain

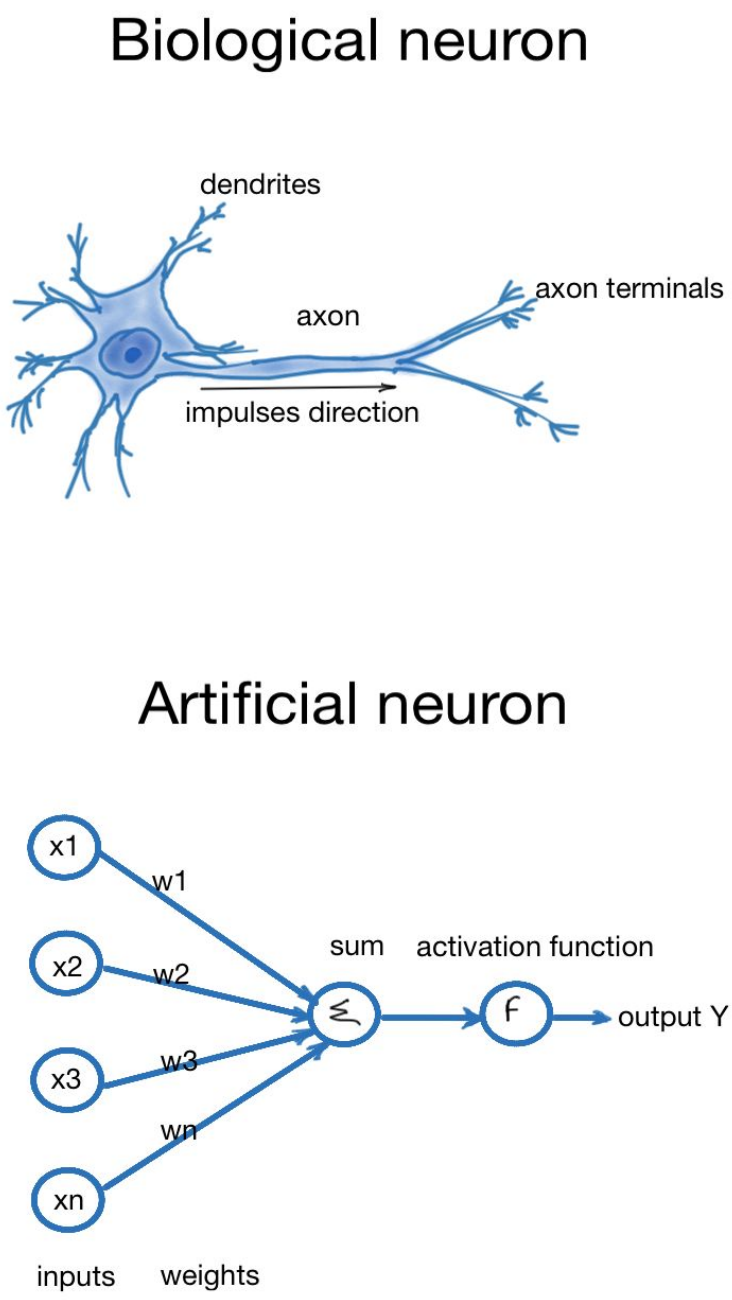


Figure 3: *
From biological neurons to artificial neurons: the architecture of inspiration

The leap from logic to learning required more than equations—it required inspiration from life itself. The structure of the **human brain** became the muse.

3.1 Biological Neuron

The biological neuron is a cell that receives electrical signals through dendrites, processes them in the soma (cell body), and transmits output along its axon to other neurons.

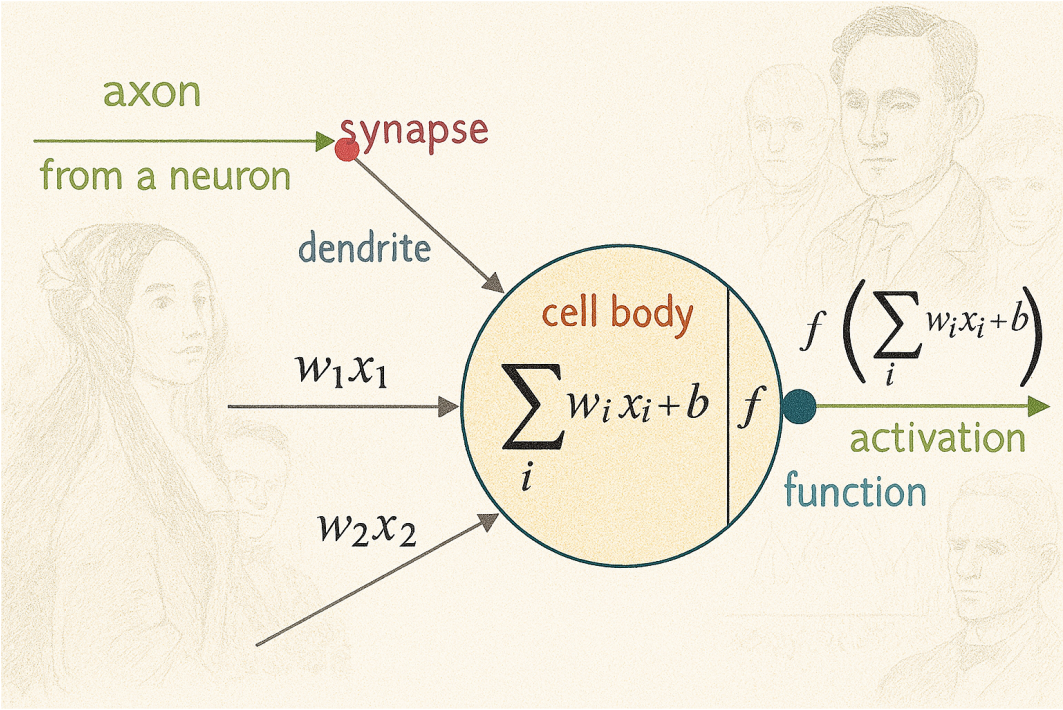


Figure 4: *
artificial neuron: mathematical model for.

Note

Neurons "fire" when their combined inputs exceed a certain threshold. This is the principle that McCulloch and Pitts mathematically modeled in 1943.

3.2 The McCulloch–Pitts Neuron: A Linear Classifier

In 1943, **Warren McCulloch** and **Walter Pitts** proposed the first mathematical model of a neuron. While biological neurons exhibit complex electrochemical dynamics, their model was beautifully simple—and paved the way for artificial neural networks.

Note

This model treats the neuron as a linear classifier: it "fires" (activates) when a linear combination of its inputs exceeds a threshold.

Each neuron j receives inputs a_i from connected neurons and applies weights w_{ij} to compute a total input:

$$\text{in}_j = \sum_{i=0}^{n} w_{ij} a_i$$

The output activation of the neuron is then computed by applying an activation function g to the weighted sum:

$$a_j = g(\text{in}_j) = g\left(\sum_{i=0}^{n} w_{ij} a_i\right)$$

Note that the input a_0 is always set to 1. This is called a **bias input**, or dummy input. Since our model is linear and we want to be able to represent constant offsets (like the intercept in a linear equation), this bias input allows the network to learn a constant term.

Example

If we omit the bias term, the neuron could only model functions that pass through the origin. The bias $a_0 = 1$ gives the network flexibility to shift the activation boundary.

Example

In nature, neurons strengthen their connections with experience. In machines, weights are updated by algorithms like gradient descent. The ideas are strikingly parallel.

Reflection

AI did not arise from code alone—it grew from the desire to emulate biology. Every node in a neural network today carries the DNA of the brain.

3.3 Activating Intelligence: The Role of the Activation Function

Neurons in a network compute weighted sums of inputs—but the magic happens when we pass that result through a function called the **activation function**. This function determines whether (and how strongly) a neuron "fires." Without it, a neural network would be nothing more than a glorified linear regression.

Why Do We Need Activation Functions?

Activation functions introduce the non-linearity that allows neural networks to model complex behaviors. Stacking linear operations without a non-linear step is futile: the result is still linear. Activation functions let us break free from that limitation.

What Makes a Good Activation Function?

An effective activation function should:

- **Be non-linear** – to capture complex, real-world relationships.
- **Be inspired by biology** – neurons don't output binary decisions only; they respond with varying intensity.
- **Be differentiable** – or almost everywhere, to enable learning using gradient-based methods like backpropagation.

Common Activation Functions

Step Function:

$$g(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Historically important and easy to understand, but not used in modern networks because it's not differentiable and provides no gradient information.

Sigmoid Function:

$$g(x) = \frac{1}{1 + e^{-kx}}$$

Smooth and differentiable, it squashes input to a range between 0 and 1. However, it suffers from the *vanishing gradient problem*—its slope becomes too flat for large inputs, making training slow or ineffective.

ReLU (Rectified Linear Unit):

$$g(x) = \max(0, x)$$

Simple, fast, and widely used. It activates only for positive inputs and outputs zero otherwise. Its downside? When inputs are negative, the gradient is zero—this can "kill" some neurons during training.

Leaky ReLU:

$$g(x) = \max(0.01x, x)$$

A clever tweak on ReLU. It allows a small, non-zero gradient when the input is negative, preventing neurons from dying during training.

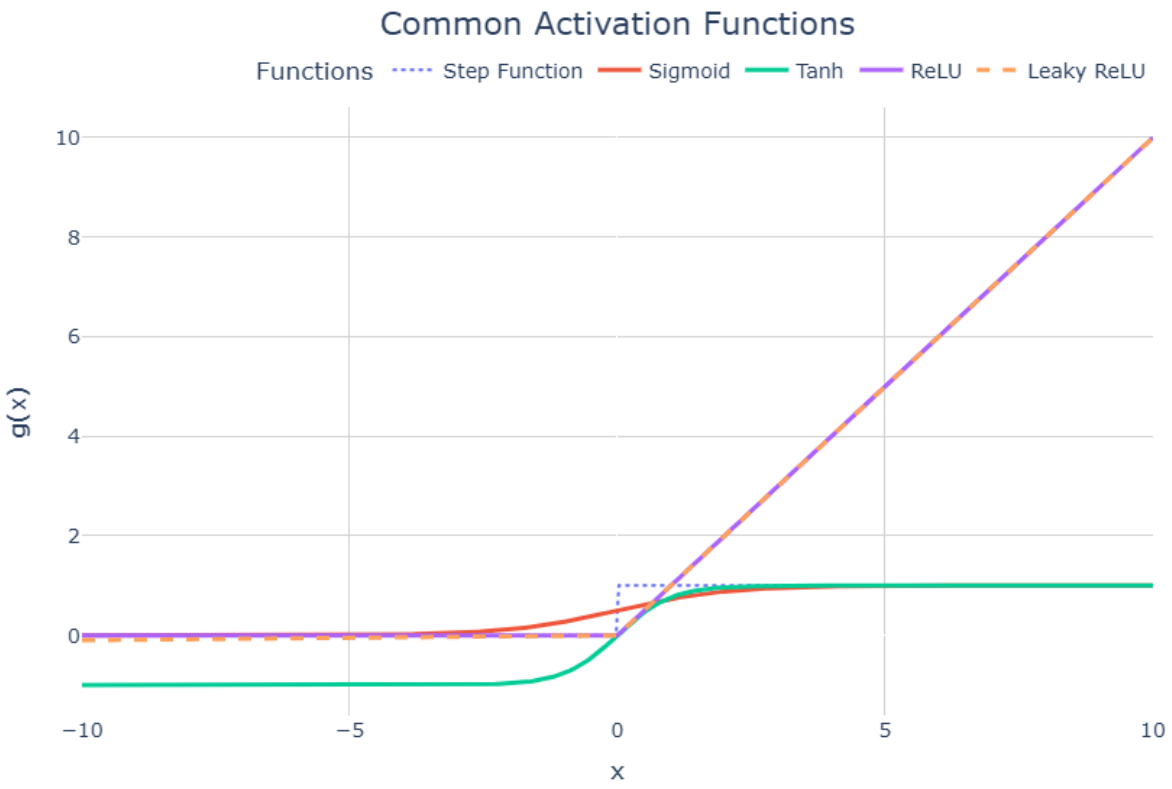


Figure 5: *
Common Activation Functions: Visual comparison of Sigmoid, Tanh, ReLU, Leaky ReLU, and Step functions.

Note

Choosing an activation function is like choosing the personality of your network. Sigmoid is calm and bounded. ReLU is bold but unforgiving. Leaky ReLU is forgiving, but slightly unpredictable.

Reflection

The activation function is the bridge between mathematics and cognition. It transforms raw computation into adaptive response—mirroring the way biological neurons react to stimulus. It's a small function with enormous influence.

4 The Perceptron: The Dawn of Machine Learning

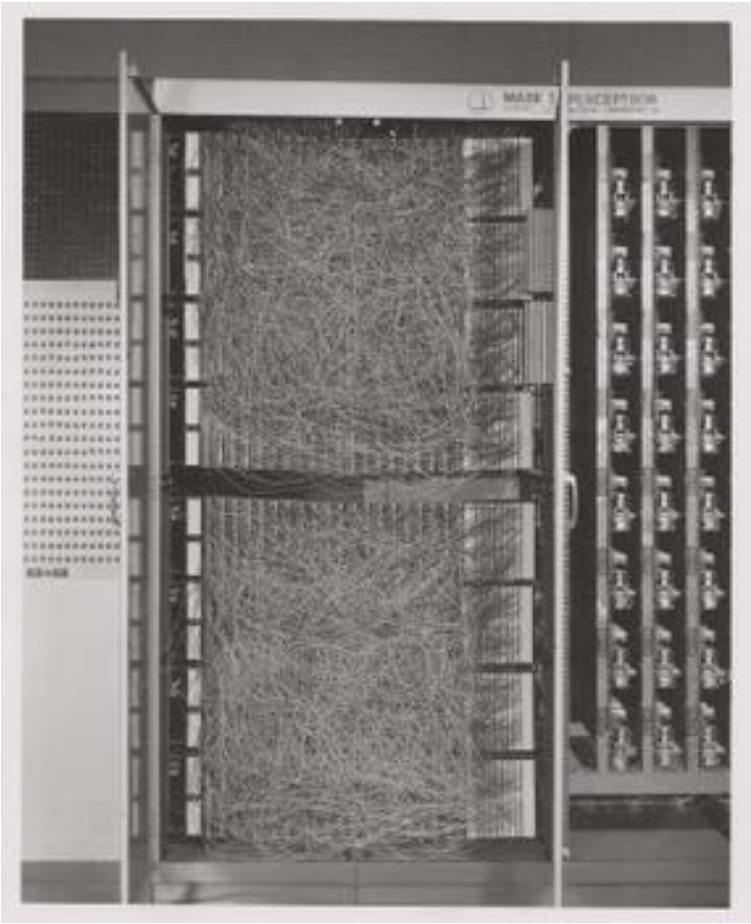


Figure 6: *
The Mark I Perceptron (Cornell, 1958) — the first neural network hardware built to learn from experience. Developed at Cornell Aeronautical Laboratory, it was capable of image recognition tasks using a simple model of interconnected neurons.

In 1958, psychologist **Frank Rosenblatt** introduced the *Perceptron*, a computational model inspired by biological neurons. It marked the birth of machine learning — a system that could, for the first time, *learn from data* rather than being explicitly programmed. The perceptron consisted of layers of artificial neurons that adjusted their weights based on experience, forming the conceptual foundation of today's deep learning networks.

Rosenblatt's prototype, the **Mark I Perceptron**, was implemented as an electromechanical machine connected to an array of photoelectric cells. It could distinguish simple patterns such as letters and shapes, foreshadowing the pattern recognition capabilities of modern AI.

The perceptron era set in motion the dream of machines that could perceive and adapt. Although its limitations (highlighted by Minsky and Papert in 1969) led to an "AI winter," its principles survived and evolved into the neural networks that power today's intelligent

Figure 7: *
Frank Rosenblatt — AI Reimagined (Generated by ChatGPT). A modern artistic tribute to Rosenblatt, the creator of the first learning algorithm. This image symbolizes how far artificial intelligence has come — from the perceptron of the 1950s to today's AI models capable of generating art, language, and scientific discovery.

systems. From Rosenblatt's hand-wired perceptron to AI-generated art, the circle of innovation has come full — a testament to how ideas born in the 1950s now define the 21st century.

4.1 The Perceptron Algorithm: A Classic Learning Rule

Among early learning algorithms, the **Perceptron** stands out—not only for its simplicity, but also for its foundational role in neural network history. Inspired by the neuron model of McCulloch and Pitts (1943) and Hebbian learning principles, the perceptron was introduced by Frank Rosenblatt in 1962 as a simple, mistake-driven learning algorithm.

Note

Originally, the perceptron was not seen as optimizing a cost function, but rather as a behavioral rule: if a neuron misfires, adjust its weights so it gets it right next time.

4.2 Learning Setup

We are given a training dataset $D_n = \{(x^{(i)}, y^{(i)})\}_{i=1}^n$, where each $x^{(i)} \in \mathbb{R}^d$ is a feature vector and $y^{(i)} \in \{-1, +1\}$ is its corresponding label.

The perceptron computes a weighted sum of inputs:

$$\text{in} = w^\top x + b = \sum_{j=1}^{d} w_j x_j + b$$

and predicts:

$$h(x) = \text{sign}(w^\top x + b)$$

—

Perceptron Learning Algorithm

Goal: Learn a linear classifier $h(x) = \text{sign}(w^\top x + b)$
Initialize: $w = 0 \in \mathbb{R}^d$, $b = 0$
For each training example $(x^{(i)}, y^{(i)})$:

- If $y^{(i)}(w^\top x^{(i)} + b) \leq 0$:

$$w \leftarrow w + y^{(i)} x^{(i)}$$
$$b \leftarrow b + y^{(i)}$$

Return: Learned weights w and bias b

Example

Each time the perceptron misclassifies an input, it adjusts the weights in the direction of the correct label. If the example was positive but predicted negative, the weight vector is pulled toward the input. If the example was negative and misclassified, the weight is pushed away.

Convergence and Behavior

The perceptron only updates weights when it makes a mistake. If a full pass through the dataset results in no updates, the algorithm halts—it has found a consistent linear separator.

Reflection

The perceptron may be simple, but it introduced core ideas like online learning, mistake-driven updates, and linear decision boundaries. Its legacy lives on in modern classifiers like support vector machines and deep networks.

Example

The Mark I could recognize shapes like triangles or vertical lines. It was limited—but it proved that machines could learn.

Note

Inspired by the brain, and rooted in Turing's theory of machines, the perceptron was the moment when "thinking machines" moved from idea to implementation.

Reflection

The perceptron was not just hardware—it was a declaration. It said: *Machines can be taught.* From this seed, all modern neural networks would grow.

4.3 Perceptrons as Logic Gates

A surprising yet elegant result: a single perceptron can replicate simple logical functions like **AND**, **OR**, and **NOT**. This was revolutionary—at a time when AI focused on symbolic logic and deduction, it showed that neurons could do logical reasoning with only real-valued weights and a step function.

Let's examine a concrete example.

Example

Problem:
Consider the following perceptron where the activation function is the step function:

$$g(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

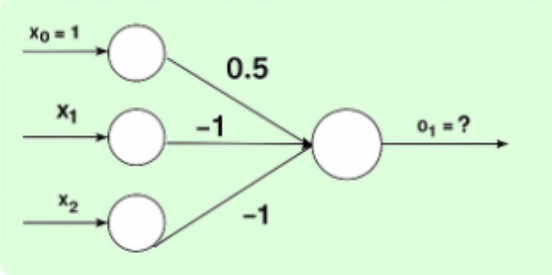
Which of the following logical functions does the perceptron compute?

(A) $x_1 \wedge x_2$

(B) $\neg(x_1 \wedge x_2)$

(C) $x_1 \vee x_2$

(D) $\neg(x_1 \vee x_2)$



Explanation

Let's analyze what this perceptron does.
The net input is computed as:

$$s = 0.5 \cdot x_0 + (-1) \cdot x_1 + (-1) \cdot x_2$$

Since $x_0 = 1$ (bias), this becomes:

$$s = 0.5 - x_1 - x_2$$

The perceptron fires ($o_1 = 1$) only if $s > 0$, that is, when both x_1 and x_2 are 0. All other input combinations result in $s \leq 0$ and thus $o_1 = 0$.

This truth table confirms it:

x_1	x_2	Output
0	0	1
0	1	0
1	0	0
1	1	0

The output is **true only when both inputs are false**, which is equivalent to:

$$o_1 = \neg(x_1 \vee x_2)$$

Thus, the correct answer is **(D)**.

4.4 The Limits of Perceptrons: The Case of XOR

In the golden years of early neural networks, perceptrons held great promise. They could simulate logic gates like AND, OR, and NOT—building blocks of logical reasoning. But that illusion of completeness was shattered by a surprising counterexample: the XOR function.

A Hidden Limitation

The perceptron is a linear classifier. Its decision boundary is a straight line (or a hyperplane in higher dimensions). But some logical functions, such as XOR, are not linearly separable.

XOR: The Unsolvable Puzzle

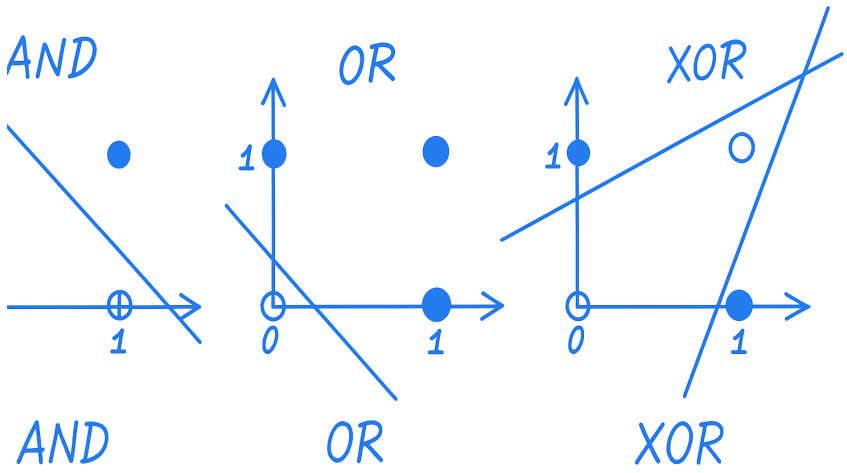


Figure 8: *
XOR is not linearly separable. No straight line can separate the 1's from the 0's.

Let's recall the XOR (exclusive OR) truth table:

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

If you plot these points in 2D space, you'll find that there's no way to draw a single straight line that separates the "1" outputs from the "0"s. This is the key insight: **no linear boundary can represent XOR**.

Proof by Contradiction

Assume a perceptron with the step activation function:

$$g(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

can compute XOR. Let the input be x_1, x_2 with a bias term $x_0 = 1$. The perceptron computes:

$$s = w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2$$

Then the output is $g(s)$. We know from XOR's truth table what outputs must be produced. This gives us:

$$\begin{aligned} w_1 \cdot 0 + w_2 \cdot 0 + w_0 \leq 0 \quad & \text{(want 0)} \\ w_1 \cdot 0 + w_2 \cdot 1 + w_0 > 0 \quad & \text{(want 1)} \\ w_1 \cdot 1 + w_2 \cdot 0 + w_0 > 0 \quad & \text{(want 1)} \\ w_1 \cdot 1 + w_2 \cdot 1 + w_0 \leq 0 \quad & \text{(want 0)} \end{aligned}$$

Simplifying:

$$\begin{aligned} w_0 &\leq 0 \\ w_2 + w_0 > 0 &\Rightarrow w_2 > -w_0 \\ w_1 + w_0 > 0 &\Rightarrow w_1 > -w_0 \\ w_1 + w_2 + w_0 &\leq 0 \end{aligned}$$

But adding the second and third inequalities gives:

$$w_1 + w_2 > -2w_0$$

So:

$$w_1 + w_2 + w_0 > -2w_0 + w_0 = -w_0 \Rightarrow w_1 + w_2 + w_0 > -w_0 \Rightarrow w_1 + w_2 + w_0 > 0$$

Which contradicts the fourth inequality $w_1 + w_2 + w_0 \leq 0$. Thus, no such weights can exist.

Note

Conclusion: A single-layer perceptron cannot learn XOR. This is not just a technicality—it led to the first "AI Winter" in the 1970s when research funding for neural networks dropped dramatically.

4.5 The Way Forward

The solution came decades later: **multi-layer networks**. By stacking perceptrons and introducing non-linear activation functions, XOR and other non-linearly separable functions could finally be learned.

Reflection

Early AI researchers believed that if perceptrons could compute logic, they could compute reasoning. This turned out to be partly true—simple perceptrons are limited—but the dream of learning machines that *think* started here.

5 Neural Networks: From Layers to Learning

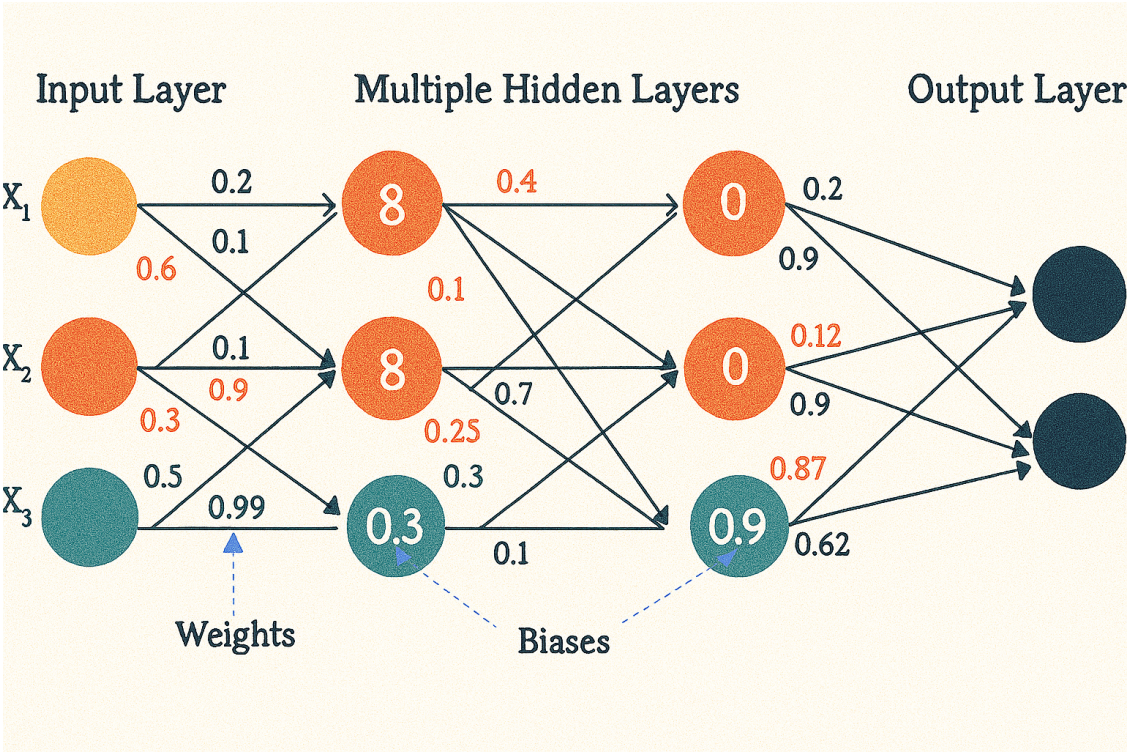


Figure 9: *
A typical neural network architecture: input layer, hidden layers, and output layer

Note

Neural networks are built from layers of interconnected nodes—each performing simple computations, but together enabling complex pattern recognition.

A **neural network** consists of multiple layers:

- **Input layer:** Receives raw data (e.g., pixels, features).
- **Hidden layers:** Each node (or neuron) computes a weighted sum of its inputs, applies an activation function, and passes the result forward.
- **Output layer:** Produces the final prediction (classification, score, etc.).

Each node (or *artificial neuron*) mimics a biological neuron by processing incoming values and deciding whether to "fire" (i.e., produce a signal) based on its activation function.

Feed-forward vs. Recurrent Neural Networks

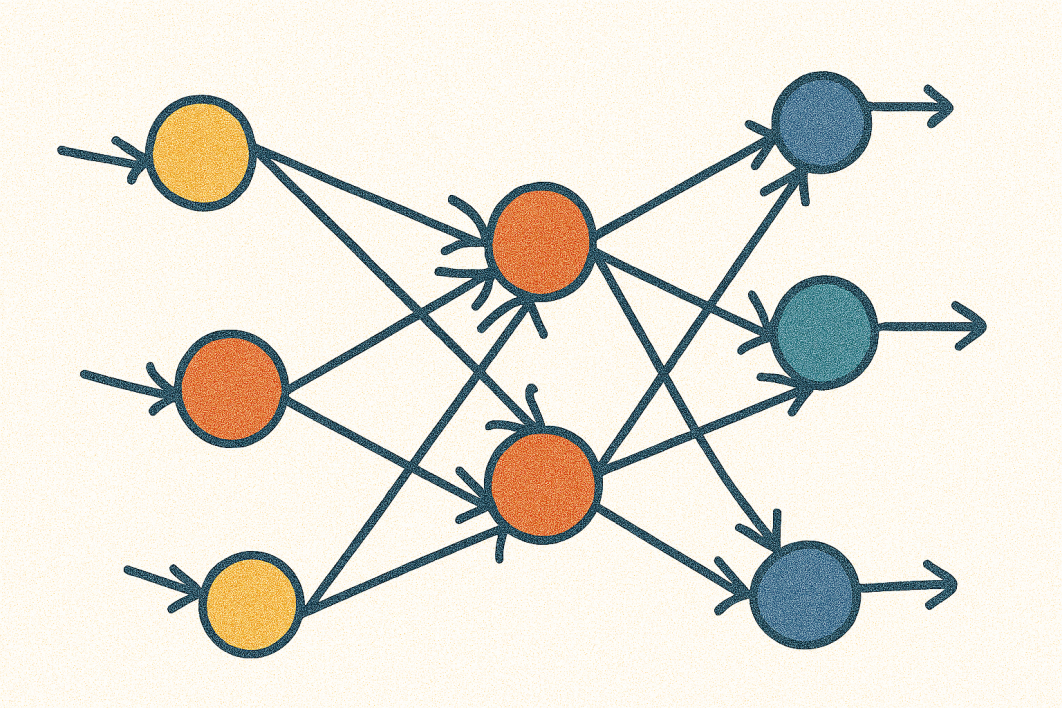


Figure 10: *
Feed-forward networks

Feed-forward neural networks are the most common type. Their structure is a directed acyclic graph (DAG), which means:

- Information flows in one direction—*from input to output.*
- There are no loops or cycles.
- Outputs are deterministic functions of the inputs.

Example

In a feed-forward network, a signal goes through layers and transforms step by step. Once passed, it cannot influence previous layers again.

By contrast, **recurrent neural networks (RNNs)** allow loops: the output of one node can be fed back as input. This makes RNNs useful for:

- Modeling sequences (like language or time series)
- Remembering past inputs
- Maintaining *state* over time

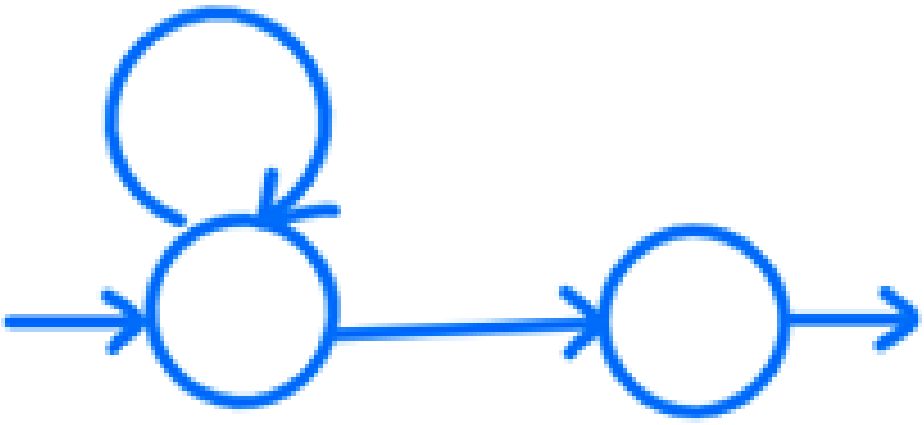


Figure 11: *
recurrent networks (right)

However, RNNs are harder to train, and their behavior is more complex to interpret. While they more closely mimic aspects of human cognition (like memory), this added power comes with computational cost.

Takeaway

In this course, we'll focus on feed-forward networks. They are the foundation for understanding deeper architectures and have been historically important in the development of AI.

5.1 Learning from Mistakes: Backpropagation

To train a neural network, we must adjust its weights to reduce prediction error. But how? The answer lies in **backpropagation**, a clever algorithm that uses calculus to compute how much each weight in the network contributed to the error—so it can be corrected.

What is Backpropagation?

Backpropagation is short for "backward propagation of errors." It tells each neuron how it should update its weights to minimize a loss function (like mean squared error). At the heart of this process lies a fundamental idea from calculus: the **chain rule**.

Note

If a function is built by chaining many smaller differentiable operations (like addition, multiplication, activation functions), then we can also compute its derivative by chaining the derivatives of those smaller parts.

A Quick Refresher: The Chain Rule

We begin by recalling the chain rule in calculus. Suppose a scalar variable J depends on intermediate variables $g_1, g_2, \ldots, g_k$, which themselves depend on input variables $x_1, x_2, \ldots, x_p$. Formally, we can express this as:

$$g_j = g_j(x_1, \ldots, x_p), \quad \forall j \in \{1, \ldots, k\}$$

$$J = J(g_1, \ldots, g_k)$$

Then, by the chain rule, for any $i \in \{1, \ldots, p\}$,

$$\frac{\partial J}{\partial x_i} = \sum_{j=1}^{k} \frac{\partial J}{\partial g_j} \frac{\partial g_j}{\partial x_i}$$

This result tells us how the change in J with respect to an input x_i depends on all intermediate variables g_j that are functions of x_i.

In the context of **neural networks**, this principle is exactly what **backpropagation** uses. Each layer computes gradients with respect to its inputs by applying the chain rule recursively—

This elegant rule allows neural networks to efficiently update their weights and learn from data.

5.2 A Simple Example: One Neuron

Imagine a neuron that takes an input x, multiplies it by a weight w, adds a bias b, and passes the result through an activation function (like ReLU). It outputs o, and we compare o to the target y with a loss function:

$$z = w \cdot x + b, \quad o = \text{ReLU}(z), \quad J = \frac{1}{2}(y - o)^2$$

We want to update w and b to reduce J. Using the chain rule:

$$\frac{dJ}{dw} = \frac{dJ}{do} \cdot \frac{do}{dz} \cdot \frac{dz}{dw}$$

Each part is simple:

$$\frac{dJ}{do} = o - y, \quad \frac{do}{dz} = \text{ReLU}'(z), \quad \frac{dz}{dw} = x$$

Putting it all together:

$$\frac{dJ}{dw} = (o - y) \cdot \text{ReLU}'(z) \cdot x$$

Similarly, we compute the gradient w.r.t b by

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial o} \cdot \frac{\partial o}{\partial z} \cdot \frac{\partial z}{\partial b} = (o - y) \cdot \text{ReLU}'(z)$$

(because $\frac{\partial o}{\partial b} = (o - y)$ and $\frac{\partial z}{\partial z} = \text{ReLU}'(z)$ and $\frac{\partial z}{\partial b} = 1$)

Then we update weights like:

$$w \leftarrow w - \eta \cdot \frac{\partial J}{\partial w}$$

$$b \leftarrow b - \eta \cdot \frac{\partial J}{\partial b}$$

where η is the learning rate.

Two-layer Neural Network

For a small network with one hidden layer:

$$a = \text{ReLU}(W^{[1]}x + b^{[1]}), \quad o = W^{[2]}a + b^{[2]}, \quad J = \frac{1}{2}(y - o)^2$$

We start by computing the output, then propagate the error backwards:

$$\delta^{[2]} = (o - y)$$

$$\delta^{[1]} = (W^{[2]})^T \delta^{[2]} \cdot \text{ReLU}'(z^{[1]})$$

And use these deltas to update the weights:

$$\frac{dJ}{W^{[2]}} = \delta^{[2]} \cdot a^T, \quad \frac{dJ}{W^{[1]}} = \delta^{[1]} \cdot x^T$$

Takeaway

Backpropagation teaches the network how to learn by computing how each layer contributes to the final error. It's efficient, elegant, and essential.

5.3 Two-layer neural networks: an intuitive step-by-step derivation

Note: this subsection derives the derivatives with low-level notations to help you build up intuition on backpropagation. If you are looking for a clean formula, or you are familiar with matrix derivatives, then feel free to jump to the next subsection directly.

Now we consider the two-layer neural network defined in equation (7.11). We compute the loss J by following sequence of operations

$$\begin{aligned} \forall j \in [1, ..., m], \quad z_j &= w_j^{[1]\top} x + b_j^{[1]} \text{ where } w_j^{[1]} \in \mathbb{R}^d, b_j^{[1]} \in \mathbb{R} \\ a_j &= \mathrm{ReLU}(z_j), \\ a &= [a_1, \ldots, a_m]^\top \in \mathbb{R}^m \\ o &= w^{[2]\top} a + b^{[2]} \text{ where } w^{[2]} \in \mathbb{R}^m, b^{[2]} \in \mathbb{R} \\ J &= \frac{1}{2}(y - o)^2 \end{aligned} \tag{7.34}$$

We will use $(w^{[\ell]})_t$ to denote the t-th coordinate of $w^{[\ell]}$, and $(w_j^{[\ell]})_t$ to denote the ℓ-coordinate of $w_j^{[\ell]}$. (We will avoid using these cumbersome notations once we figure out how to write everything in matrix and vector forms.)

By invoking chain rule with J as the output variable, o as intermediate variable, and $(w^{[2]})_\ell$ as the input variable, we have

$$\begin{aligned} \frac{\partial J}{\partial (w^{[2]})_\ell} &= \frac{\partial J}{\partial o} \cdot \frac{\partial o}{\partial (w^{[2]})_\ell} \\ &= (o - y) \frac{\partial o}{\partial (w^{[2]})_\ell} \\ &= (o - y) a_\ell \end{aligned}$$

It's more challenging to compute $\frac{\partial J}{\partial (w_j^{[1]})_\ell}$. Towards computing it, we first invoke the chain rule with J as the output variable, z_j as the intermediate variable, and $(w_j^{[1]})_\ell$ as the input variable.

$$\begin{aligned}\frac{\partial J}{\partial (w_j^{[1]})_\ell} &= \frac{\partial J}{\partial z_j} \cdot \frac{\partial z_j}{\partial (w_j^{[1]})_\ell} \\ &= \frac{\partial J}{\partial z_j} \cdot x_\ell \qquad (\text{because } \frac{\partial z_j}{\partial x_\ell} = x_\ell)\end{aligned}$$

Thus, it suffices to compute the $\frac{\partial J}{\partial z_j}$. We invoke the chain rule with J as the output variable, a_j as the intermediate variable, and z_j as the input variable,

$$\begin{aligned}\frac{\partial J}{\partial z_j} &= \frac{\partial J}{\partial a_j}\frac{\partial a_j}{\partial z_j} \\ &= \frac{\partial J}{\partial a_j}\text{ReLU}'(z_j)\end{aligned}$$

Now it suffices to compute $\frac{\partial J}{\partial a_j}$, and we invoke the chain rule with J as the output variable, o as the intermediate variable, and a_j as the input variable,

$$\begin{aligned}\frac{\partial J}{\partial a_j} &= \frac{\partial J}{\partial o}\frac{\partial o}{\partial a_j} \\ &= (o - y) \cdot (w^{[2]})_j\end{aligned}$$

Now combining the equations above, we obtain

$$\frac{\partial J}{\partial (w_j^{[1]})_\ell} = (o - y) \cdot (w^{[2]})_j \text{ReLU}'(z_j) x_\ell$$

Algorithm 1 Backpropagation for two-layer neural networks

1: Compute the values of $z_1, \ldots, z_m$, $a_1, \ldots, a_m$ and o as in the definition of neural network (equation (7.34)).
2: Compute $\frac{\partial J}{\partial o} = (o - y)$.
3: Compute $\frac{\partial J}{\partial z_j}$ for $j = 1, \ldots, m$ by

$$\frac{\partial J}{\partial z_j} = \frac{\partial J}{\partial o}\frac{\partial o}{\partial a_j}\frac{\partial a_j}{\partial z_j} = \frac{\partial J}{\partial o} \cdot (w^{[2]})_j \cdot \mathrm{ReLU}'(z_j) \tag{7.35}$$

4: Compute $\frac{\partial J}{\partial (w_j^{[1]})_\ell}$, $\frac{\partial J}{\partial b_j^{[1]}}$, $\frac{\partial J}{\partial (w^{[2]})_j}$, and $\frac{\partial J}{\partial b^{[2]}}$ by

$$\begin{aligned}
\frac{\partial J}{\partial (w_j^{[1]})_\ell} &= \frac{\partial J}{\partial z_j} \cdot \frac{\partial z_j}{\partial (w_j^{[1]})_\ell} = \frac{\partial J}{\partial z_j} \cdot x_\ell \\
\frac{\partial J}{\partial b_j^{[1]}} &= \frac{\partial J}{\partial z_j} \cdot \frac{\partial z_j}{\partial b_j^{[1]}} = \frac{\partial J}{\partial z_j} \\
\frac{\partial J}{\partial (w^{[2]})_j} &= \frac{\partial J}{\partial o} \cdot \frac{\partial o}{\partial (w^{[2]})_j} = \frac{\partial J}{\partial o} \cdot a_j \\
\frac{\partial J}{\partial b^{[2]}} &= \frac{\partial J}{\partial o}\frac{\partial o}{\partial b^{[2]}} = \frac{\partial J}{\partial o}
\end{aligned}$$

5.4 Multi-layer neural networks

In this part, we present the backpropagation steps for deep neural networks, specifically those described by:

$$\begin{aligned}
a^{[1]} &= \mathrm{ReLU}(W^{[1]}x + b^{[1]}) \\
a^{[2]} &= \mathrm{ReLU}(W^{[2]}a^{[1]} + b^{[2]}) \\
&\vdots \\
a^{[r-1]} &= \mathrm{ReLU}(W^{[r-1]}a^{[r-2]} + b^{[r-1]}) \\
a^{[r]} &= z^{[r]} = W^{[r]}a^{[r-1]} + b^{[r]} \\
J &= \frac{1}{2}(a^{[r]} - y)^2
\end{aligned}$$

Here, we simplify notation by letting both $a^{[r]}$ and $z^{[r]}$ represent the network output $h_\theta(x)$.

We define the following quantity to ease our gradient calculations:

$$\delta^{[k]} \triangleq \frac{\partial J}{\partial z^{[k]}} \tag{3.16}$$

The backpropagation process recursively computes $\delta^{[k]}$ for each layer $k = r, r-1, ..., 1$, and

then uses them to update weights via:

$$\frac{\partial J}{\partial W^{[k]}} = \delta^{[k]} a^{[k-1]\top}, \quad \frac{\partial J}{\partial b^{[k]}} = \delta^{[k]}$$

Algorithm 2 Backpropagation for multi-layer networks

1: Compute and store activations $a^{[k]}$, pre-activations $z^{[k]}$ for all layers $k = 1, \ldots, r$
2: Set $\delta^{[r]} = (z^{[r]} - y)$
3: **for** each layer $k = r - 1$ down to 1 **do**
4: Compute:

$$\delta^{[k]} = \left(W^{[k+1]\top} \delta^{[k+1]}\right) \circ \mathrm{ReLU}'(z^{[k]})$$

5: Compute gradients:

$$\frac{\partial J}{\partial W^{[k+1]}} = \delta^{[k+1]} a^{[k]\top}, \quad \frac{\partial J}{\partial b^{[k+1]}} = \delta^{[k+1]}$$

6: **end for**

5.5 Vectorization Over Training Examples

To efficiently compute outputs and gradients over a batch of training examples, we express the forward and backward passes in matrix form.

Example. Assume three examples $x^{(1)}, x^{(2)}, x^{(3)}$. For the first layer:

$$z = W^{[1]} x^{(1)} + b^{[1]}$$
$$z = W^{[1]} x^{(2)} + b^{[1]}$$
$$z = W^{[1]} x^{(3)} + b^{[1]}$$

To vectorize this computation, define:

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} \end{bmatrix} \in \mathbb{R}^{d \times 3}$$

$$Z^{[1]} = W^{[1]} X + b^{[1]} \tag{4.2}$$

Here, $b^{[1]} \in \mathbb{R}^{m \times 1}$ is broadcast across all columns.

Note. In practical deep learning frameworks (e.g., PyTorch, TensorFlow), data points are stored row-wise. So the input data matrix is $X \in \mathbb{R}^{3\times d}$, and the computations become:

$$Z^{[1]} = XW^{[1]} + b^{[1]} \in \mathbb{R}^{3\times m} \tag{4.4}$$

This ensures compatibility with the matrix dimensions during training.

References

[1] W. S. McCulloch and W. Pitts. *A Logical Calculus of the Ideas Immanent in Nervous Activity*. Bulletin of Mathematical Biophysics, vol. 5, pp. 115–133, 1943.

[2] Frank Rosenblatt. *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain*. Psychological Review, vol. 65, no. 6, pp. 386–408, 1958.

[3] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Representations by Back-Propagating Errors*. Nature, vol. 323, pp. 533–536, 1986.

[4] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. *A Fast Learning Algorithm for Deep Belief Nets*. Neural Computation, vol. 18, no. 7, pp. 1527–1554, 2006.

[5] Alan M. Turing. *Computing Machinery and Intelligence*. Mind, vol. 59, no. 236, pp. 433–460, 1950.

[6] Alice Gao. *Neural Networks Lecture Series*. Online lecture notes and videos, 2020. Available at: https://csc413-w25.notion.site/

[7] Andrew Ng. *CS229: Machine Learning Lecture Notes*. Stanford University, 2018.