# WTFunk is Deep Learning

A Guide for Self-Learners

*"The most difficult thing is the decision to act; the rest is merely tenacity."*
– Amelia Earhart

**Written by:** Pierre Chambet
**Date:** June 2, 2025

---

*This guide was crafted with patience, clarity, and a healthy dose of curiosity.*
*If you stick with it, it may just change how you see deep learning.*

# WTFunk is Deep Learning

A Guide for Self-Learners

> *"The most difficult thing is the decision to act; the rest is merely tenacity."*
> *– Amelia Earhart*

Hello. Today, you stand before this numerical sheet of paper, and I see a person who wants things to change. But most importantly, I see someone ready to take full responsibility for their actions. That's sometimes rare - but who said that pearls are easy to find?

Welcome to my deep learning guide, written by someone who has asked himself the same questions you're probably asking right now:

*Wtfunk is Deep Learning?*
*How does that work...? Like REALLY?*

This notebook is hard. But it's also a journey - a journey that will take you from the very basics of deep learning to a fully operational multi-layer neural network.

But don't worry. I'm not a preacher saying that - I mean it. Why? Because I'm not a teacher. **I'm like you.**

And most importantly: don't worry because the hardest part is the decision to act. And you just took it.

*The rest is merely tenacity.*
***So be tenacious - this notebook starts now.***

## Structure of this notebook

This notebook is composed of two parts:

- **Theoretical part** - The core concepts behind deep learning. Everything is explained in a simple, intuitive way, with illustrations and guided reasoning.
- **Practical part** - A hands-on implementation of those concepts. You'll code your own neural networks step-by-step, understand every line, and see how theory becomes practice.

That's it.

The next step for you is `theory_00.ipynb`, page **3**.

Go for it - you'll be amazed by what you can learn.

*If you still want to waste your time, you can read this.*
*Otherwise, turn to the next page.*

## Notebook Roadmap

**Understanding the neuron**

In this part, we debunk the myth of the neuron and understand how it works. Concepts are explained simply, with examples and illustrations.
**Page 3** — `theory_00.ipynb`

**Can it predict a toxic/non-toxic plant?**

We code the same neuron to predict whether a plant is toxic or not, and use it to make simple predictions.
**Page 17** — `practice_00.ipynb`

**Cat vs Dog**

Here we use the same neuron again (this little man works too much) to classify images of cats and dogs and make predictions.
**Page 32** — `practice_01.ipynb`

**Understanding the neural network**

Now we debunk the myth of the neural network and understand how it works. Simple, straight to the point.
**Page 44** — `theory_02.ipynb`

**Coding a two-layer neural network**

We code a first neural network with two layers and see how it handles the previous datasets.
**Page 68** — `practice_02.ipynb`

**Coding a multi-layer neural network**

They all say it's complex. But deep down, it's just fear talking. If you've followed along this far, you'll be blown away by how simple it really is.
**Page 95** — `practice_03.ipynb`

## What is a Neural Network?

A neural network is... a network of neurons. *I'm not joking.* A neural network is like a team of decision-makers (neurons), each making a tiny choice based on inputs. Combined, they learn complex patterns - like how you recognize faces or letters.

If you understand how a neuron works, you'll understand how a network of neurons works. So let's first understand how a single neuron works, before understanding how a network of neurons works.

**A neuron is just a linear function.**

At the beginning of your AI journey, when building your first deep learning models, many of your questions can be answered by remembering:

**A neuron is a linear function. Always keep that in mind.**

However, **how** the neuron works is slightly different - and that's where confusion often comes from.

So here's the second most important thing to know about AI:

**How a deep learning algorithm (from neuron to neural networks) works is always the same. Always.**

Here's the architecture:

1. A linear function that processes the input data
2. An activation function
3. A performance measurement
4. An optimization process to improve performance

This structure is always the same for every deep learning algorithm: **it is true for a single neuron, it is true for a neural network.**

What changes is the content: the choice of mathematical models for (1), (2), (3), and (4). The methodology of the recipe is always the same. What changes is the ingredients you decide to use.

To really understand this, let's walk through an example.

## Example: A Simple Binary Classification Problem

We will now explore how this architecture works using a binary classification model.
Don't be afraid of the terminology - it will all make sense soon.

This model allows us to linearly separate two classes.
Suppose we are studying two types of plants:

- Toxic plants (labeled $y = 1$, class 1)
- Non-toxic plants (labeled $y = 0$, class 0)

We decide to measure two features:

- $x_1$: the length of the leaves
- $x_2$: the width of the leaves

So each plant is represented as a data point $x = (x_1, x_2)$.

For example, here's a dataset we could get when we plot these measurements:

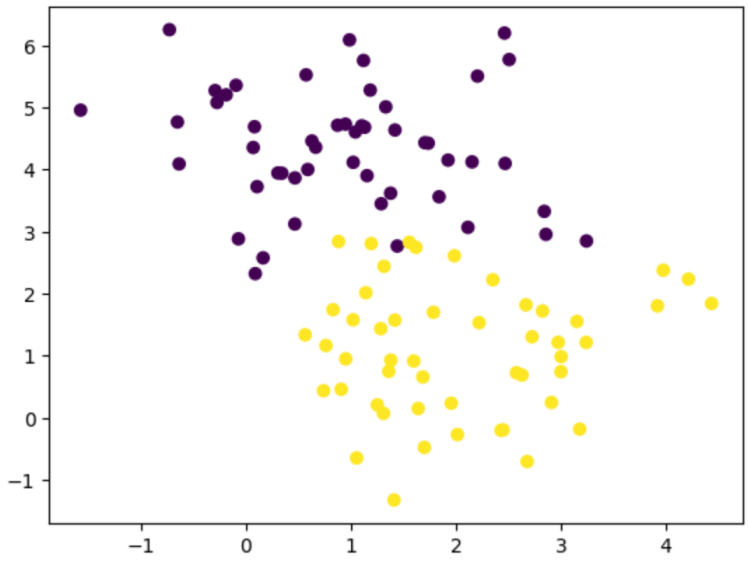



Figure 1: Random Dataset - Leaf Width vs. Length

*Purple = toxic (class 1), Yellow = non-toxic (class 0).*

To find the equation of the **decision boundary** - the line that separates toxic from non-toxic plants - we will build a deep learning algorithm: a **neuron**.

This basic unit was introduced by **Frank Rosenblatt** in *1958*. Let's now understand how it works.

## 1. Process the Input Data: A Linear Function

Each input $x$ (a plant represented as a dot) is multiplied by a weight $w$, and we add a bias term $b$. This forms the linear model - the mathematical heart of the neuron.

Since each input has two features - length and width - we define weights for each:

$$w = (w_1, w_2)$$

and one bias term $b$ that acts globally across all inputs.

Here's how a neuron applies this transformation:

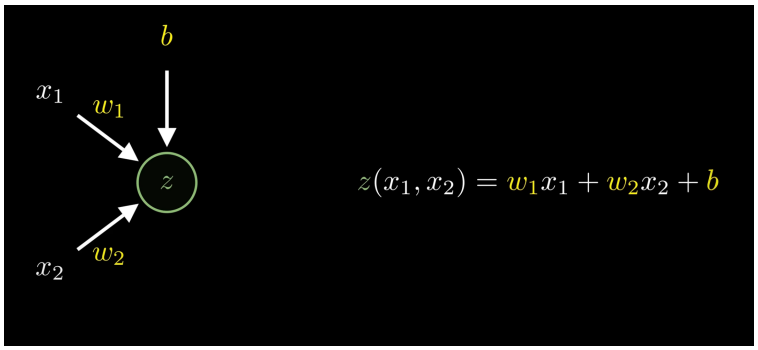



Figure 2: A neuron with its transformation function

*Now you see the linear function on each feature?*

### How Does This Neuron Work?

The output of this neuron is a value $z$, computed as:

$$z = w_1 \cdot x_1 + w_2 \cdot x_2 + b$$

This value $z$ is the raw score, the result of applying the linear model to the input.

But to make a classification, we need to interpret this value. What does $z$ mean? How can we use it to decide the correct class?

That's where the next step - the **activation function** - comes in.

## 2. Adding a Sense of Probability: Activation Functions

To make sense of the output $z$, we associate it with a **probability** - a measure of confidence in the prediction.
The further a plant is from the decision boundary, the more confident the model should be.

We use an **activation function** - a mathematical rule that "activates" the neuron - and transforms its raw score $z$ into a useful output.

A classic and intuitive choice is the **sigmoid function**, which maps any real-valued number into the interval $[0, 1]$:

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

Where $z$ is the linear output of the neuron, and $e$ is the exponential function.

This function has a natural probabilistic interpretation:

- If $a(z) > 0.5$, we predict **Class 1** (toxic)
- If $a(z) < 0.5$, we predict **Class 0** (non-toxic)

This creates a **binary classification model** - one that not only predicts a class, but also outputs how *confident* it is in that decision.

Other activation functions also exist - like **ReLU** or **tanh** - but the core idea remains the same: Apply a non-linear transformation to $z$ to give the neuron expressive power.

*Activation functions like sigmoid help neurons express "how confident" they are - not just yes/no, but values between 0 and 1.*

### Graph of the Sigmoid Function

Let's visualize the sigmoid function and highlight a specific point:

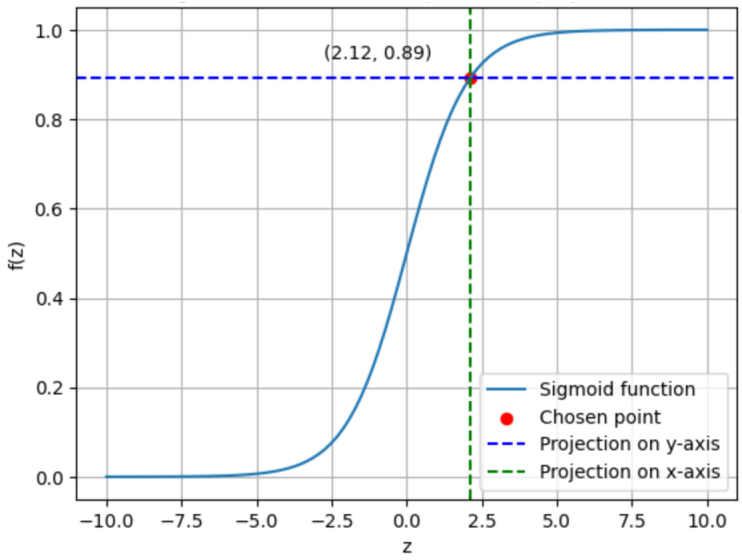



Figure 3: The sigmoid function: mapping raw scores to probabilities

*The sigmoid curve starts near 0, rises smoothly, and levels off near 1. Perfect for converting raw scores into probabilities.*

### Interpretation

Suppose the neuron output is $z = 2$. Applying the sigmoid function gives:

$$a(2) = \frac{1}{1 + e^{-2}} \approx 0.89$$

This means the model predicts an 89% chance that the plant is toxic (Class 1). That's a confident prediction - the plant lies deep in the toxic region of the feature space.

So we can interpret the output of the neuron as:

- **Probability of being toxic (class 1)**: $a(z)$
- **Probability of being non-toxic (class 0)**: $1 - a(z)$

$$P(Y = 1) = a(z), \qquad P(Y = 0) = 1 - a(z)$$

### A Bernoulli Distribution!

Each plant's classification follows a **Bernoulli distribution** with parameter $a(z)$:

$$P(Y = y) = a(z)^y \cdot (1 - a(z))^{1-y}, \quad \text{where } y \in \{0, 1\}$$

*A Bernoulli distribution models binary outcomes -*
*like flipping a coin, or deciding toxic vs. non-toxic.*

Let's verify this with the two possible outcomes:

- If $y = 1$, then:
$$P(Y = 1) = a(z)^1 \cdot (1 - a(z))^0 = a(z)$$
- If $y = 0$, then:
$$P(Y = 0) = a(z)^0 \cdot (1 - a(z))^1 = 1 - a(z)$$

So we see that the output of our neuron - interpreted via the sigmoid - follows a Bernoulli distribution with success probability $a(z)$.

*The sigmoid gives us a probability.*
*The Bernoulli distribution gives us a model to evaluate binary outcomes.*

Now we have a model that **processes input and outputs a probability**.
But how do we judge its output? How do we know if the neuron did a *good* or *bad* job?

In other words:

*How do we reward or punish our model during training?*

## 3. Performance Measurement

Now that we have a working model, we must ask: *How good is it at making correct predictions?*

To evaluate performance, we need to adjust the parameters $w$ and $b$ to minimize the difference between the model's output and the true labels.

To do that, we define a function that quantifies the discrepancy between predicted probabilities and actual outcomes - this is called the **likelihood**.

> *If a plant is toxic and the model predicts 89% probability, then the model is 89% plausible - for that example.*

The total likelihood of the model is the product of these plausibilities over all data points:

$$L(W, b) = \prod_{i=1}^{N} P(Y = y_i)$$

> *This is the product of the probability that each plant (from 1 to $N$) belongs to its true class.*

Since each prediction follows a Bernoulli distribution, we write:

$$L(W, b) = \prod_{i=1}^{N} a(z_i)^{y_i} \cdot (1 - a(z_i))^{1-y_i}$$

Where:

- $N$ is the number of plants
- $y_i$ is the true label for plant $i$
- $z_i$ is the output of the model for plant $i$

> *But we have a problem - do you see it?*

### Problem: Likelihood Gets Tiny!

Multiplying many probabilities between 0 and 1 leads to a value that quickly tends to zero:

$$\lim_{N \to \infty} L(W, b) = 0$$

This is numerically unstable for computers.

*Multiplying many small numbers leads to underflow.*
*That's a serious problem for numerical stability.*

### Solution: Shift to Logarithm

To solve the instability issue, we take the logarithm of the likelihood. Logarithms convert products into sums - which are more stable to compute:

$$\log L(W, b) = \log \left( \prod_{i=1}^{N} P(Y = y_i) \right) = \sum_{i=1}^{N} \log P(Y = y_i)$$

This gives us the **log-likelihood function**:

$$\mathcal{L}(W, b) = \sum_{i=1}^{N} [y_i \cdot \log a(z_i) + (1 - y_i) \cdot \log(1 - a(z_i))]$$

*Taking the logarithm of probabilities makes things more stable,*
*and much easier to optimize.*

### Defining the Loss Function

In machine learning, we usually prefer to **minimize a loss** rather than maximize a likelihood. So we take the negative of the log-likelihood, and normalize by the number of samples:

$$\text{Log-loss} = -\frac{1}{N} \sum_{i=1}^{N} [y_i \cdot \log a(z_i) + (1 - y_i) \cdot \log(1 - a(z_i))]$$

Where:

- $N$ is the number of plants
- $y_i$ is the true label for example $i$
- $a(z_i)$ is the predicted probability for example $i$

*The log-loss punishes confident but wrong predictions more heavily.*
*The better your confidence matches reality, the lower the loss.*

## 4. Gradient Descent

To optimize our model, we use the famous algorithm: **Gradient Descent**.

The idea is simple: We compute how the loss function changes when we slightly modify the parameters $w$ and $b$, then update them in the direction that *reduces* the loss.

We compute the gradients:

$$\frac{\partial L}{\partial w}, \qquad \frac{\partial L}{\partial b}$$

Then we apply the update rules:

$$w \leftarrow w - \alpha \cdot \frac{\partial L}{\partial w} \qquad \text{and} \qquad b \leftarrow b - \alpha \cdot \frac{\partial L}{\partial b}$$

Where $\alpha$ is the **learning rate** - a small number that controls how big each step is.

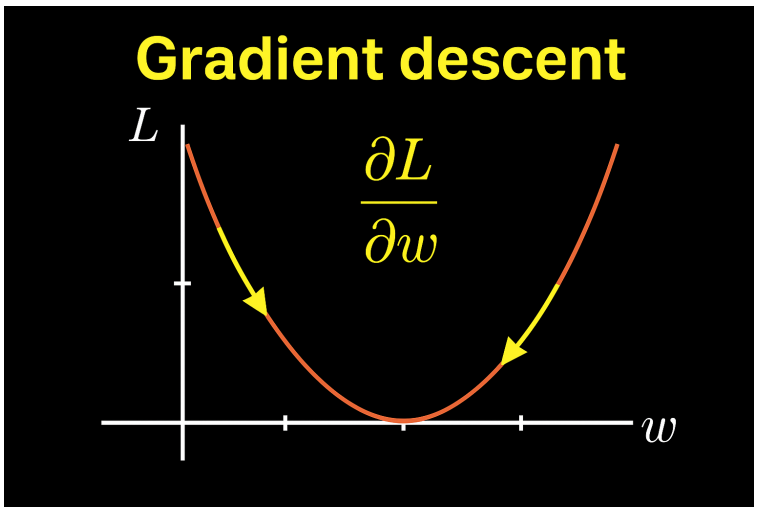



Figure 4: Finding the minimum of the loss function (2D representation)

### You Completed a Deep Learning Algorithm

Congratulations - you've just built your first deep learning model from scratch. And here's what it looks like:

1. A **linear model** to process the input features
2. An **activation function** (sigmoid) to convert raw output into probabilities
3. A **loss function** (log-loss) to evaluate prediction quality
4. An **optimization algorithm** (gradient descent) to train the model

*That's all you need to define a neuron.*
*And that's the foundation of deep learning.*

### Wait... Really?

Most courses stop here and give you the gradient formula like magic. But you want to understand - not just repeat.

Let's compute the gradients ourselves.
It's math. Not always fun. But necessary.

## Time to Listen Carefully

We're going to compute:

$$\frac{\partial L}{\partial w_1}, \qquad \frac{\partial L}{\partial w_2}, \qquad \frac{\partial L}{\partial b}$$

To do this, we'll use the powerful tool of calculus: the **chain rule**.

*The chain rule allows us to break down complex derivatives into smaller, simpler parts.*

If we have a composed function like $f(g(x))$, we differentiate it as:

$$\frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx}$$

That's exactly what we're going to do.

We'll apply the chain rule to our loss function to get the partial derivatives with respect to each parameter:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_1}$$
$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_2}$$
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial b}$$

*Each gradient is just a product of three simpler gradients.*
*Let's compute each one, step by step.*

- **$\frac{\partial L}{\partial w_1}$ computation**

Let's begin with the first parameter: we want to compute the following gradient:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_1}$$

At first glance, this may seem intimidating - but it's not. We already know the relationships between these quantities.

**Recall the structure of our neuron:**

- **Linear function:** $z = w_1 \cdot x_1 + w_2 \cdot x_2 + b$
- **Activation function:** $a(z) = \frac{1}{1+e^{-z}}$
- **Loss function:**

$$\mathcal{L}(W, b) = -\sum_{i=1}^{N} [y_i \cdot \log a(z_i) + (1 - y_i) \cdot \log(1 - a(z_i))]$$

We will now compute the small pieces step by step:

$$\boxed{\frac{\partial L}{\partial w_1} = \underbrace{\frac{\partial L}{\partial a}}_{\text{from loss}} \cdot \underbrace{\frac{\partial a}{\partial z}}_{\text{from activation}} \cdot \underbrace{\frac{\partial z}{\partial w_1}}_{\text{from linear model}}}$$

Let's compute each of these terms individually in the next sections.

**First small gradient: $\frac{\partial L}{\partial a}$**

We begin with the log-loss function:

$$\mathcal{L}(W, b) = -\sum_{i=1}^{N} [y_i \cdot \log(a(z_i)) + (1 - y_i) \cdot \log(1 - a(z_i))]$$

This function measures the error between predicted probabilities and true labels. It has two components:

- The contribution from class 1: $y_i \cdot \log(a(z_i))$
- The contribution from class 0: $(1 - y_i) \cdot \log(1 - a(z_i))$

Now, to compute the gradient with respect to $a$, we differentiate each term using the classic rule:

$$\frac{d}{dx} \log(x) = \frac{1}{x}$$

So we get:

$$\frac{\partial L}{\partial a} = -\left(\frac{y_i}{a(z)} - \frac{1 - y_i}{1 - a(z)}\right)$$

$$\boxed{\frac{\partial L}{\partial a} = -\left(\frac{y_i}{a(z)} - \frac{1 - y_i}{1 - a(z)}\right)}$$

This is the foundation: it measures how the loss changes if the prediction probability $a$ shifts slightly.

**Second small gradient: $\frac{\partial a}{\partial z}$**

The sigmoid activation function is defined as:

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

We want to compute its derivative with respect to $z$:

$$\frac{da}{dz} = \frac{d}{dz}\left(\frac{1}{1 + e^{-z}}\right)$$

We'll use the chain rule on a reciprocal function:

$$\frac{d}{dz}\left(\frac{1}{u(z)}\right) = \frac{-1}{u(z)^2} \cdot \frac{du}{dz}$$

So:

$$\begin{aligned}\frac{da}{dz} &= \frac{-1}{(1 + e^{-z})^2} \cdot \frac{d}{dz}(1 + e^{-z}) \\ &= \frac{-1}{(1 + e^{-z})^2} \cdot (-e^{-z}) \\ &= \frac{e^{-z}}{(1 + e^{-z})^2}\end{aligned}$$

Now, let's highlight a subtle and elegant **trick** - something you'll see again and again in deep learning.

We want to relate this expression to $a(z)$. Let's be clever with algebra:

$$e^{-z} = [1 + e^{-z}] - 1$$

So we write:

$$\frac{e^{-z}}{(1 + e^{-z})^2} = \frac{(1 + e^{-z} - 1)}{(1 + e^{-z})^2} = \frac{1 + e^{-z}}{(1 + e^{-z})^2} - \frac{1}{(1 + e^{-z})^2}$$

Factor out:

$$= \left(\frac{1}{1 + e^{-z}}\right) \cdot \left(1 - \frac{1}{1 + e^{-z}}\right)$$

But that's just:

$$= a(z) \cdot (1 - a(z))$$

$$\boxed{\frac{da}{dz} = a(z) \cdot (1 - a(z))}$$

This trick - rewriting $e^{-z}$ as $(1 + e^{-z}) - 1$ - is a key moment of insight. It turns a messy exponential derivative into a beautifully simple expression. One that reuses the function itself. Elegant. Efficient. Easy to remember.

### Third small gradient: $\frac{\partial z}{\partial w_1}$

Let's now compute the last term in our chain rule.

We are working with the following linear model:

$$z = w_1 \cdot x_1 + w_2 \cdot x_2 + b$$

This is the neuron's internal computation - a simple weighted sum of the inputs plus a bias.

We now differentiate $z$ with respect to $w_1$. All other terms are considered constants with respect to $w_1$:

$$\frac{\partial z}{\partial w_1} = \frac{\partial}{\partial w_1}(w_1 x_1 + w_2 x_2 + b) = x_1$$

$$\boxed{\frac{\partial z}{\partial w_1} = x_1}$$

This is the easiest of the three. The derivative of a linear term $w_1 \cdot x_1$ with respect to $w_1$ is simply $x_1$.

### Computation of $\frac{\partial \mathcal{L}}{\partial w_1}$

We now combine the gradients we computed earlier using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_1}$$

Let's go step by step:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_1} &= \left( -\frac{1}{m} \sum_i \left[ \frac{y_i}{a_i} - \frac{1 - y_i}{1 - a_i} \right] \right) \cdot a(z) \cdot (1 - a(z)) \cdot x_1 \\
&= \frac{-x_1}{m} \sum_i \left( \frac{y_i - a_i}{a_i(1 - a_i)} \cdot a_i(1 - a_i) \right) \\
&= \frac{-x_1}{m} \sum_i (y_i - a_i)
\end{aligned}$$

Gradient of $\mathcal{L}$ with respect to $w_1$

$$\boxed{\frac{\partial \mathcal{L}}{\partial w_1} = -\frac{x_1}{m} \sum_{i=1}^{m} (y_i - a_i)}$$

### Computation of $\frac{\partial \mathcal{L}}{\partial w_2}$

Now let's compute the gradient with respect to the second weight $w_2$.

We follow exactly the same logic as for $w_1$, but this time, we differentiate with respect to $w_2$:

$$z = w_1 \cdot x_1 + w_2 \cdot x_2 + b \quad \Rightarrow \quad \frac{\partial z}{\partial w_2} = x_2$$

And applying the chain rule:

$$\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_2} = \left( -\frac{1}{m} \sum_i (y_i - a_i) \right) \cdot x_2$$

**Gradient of $\mathcal{L}$ with respect to $w_2$**

$$\boxed{\frac{\partial \mathcal{L}}{\partial w_2} = -\frac{x_2}{m} \sum_{i=1}^{m} (y_i - a_i)}$$

Exactly the same structure as for $w_1$. Just replace $x_1$ with $x_2$. That's the power of symmetry in linear models.

### Computation of $\frac{\partial \mathcal{L}}{\partial b}$

Finally, we compute the gradient with respect to the bias term $b$.

From the linear model:

$$z = w_1 \cdot x_1 + w_2 \cdot x_2 + b \quad \Rightarrow \quad \frac{\partial z}{\partial b} = 1$$

Applying the same chain rule logic:

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial b} = \left( -\frac{1}{m} \sum_i (y_i - a_i) \right) \cdot 1$$

**Gradient of $\mathcal{L}$ with respect to $b$**

$$\boxed{\frac{\partial \mathcal{L}}{\partial b} = -\frac{1}{m} \sum_{i=1}^{m} (y_i - a_i)}$$

The bias is just a constant added to the weighted input - its gradient is beautifully simple.

## Update: Learning Begins

We now hold all the gradients of the loss function with respect to our parameters $w_1$, $w_2$, and $b$. That's the key. We can now **teach** our neuron by adjusting its parameters to reduce the loss. Using the gradient descent rule, we update the parameters as follows:

$$w_1 \leftarrow w_1 - \alpha \cdot \frac{\partial \mathcal{L}}{\partial w_1} \qquad w_2 \leftarrow w_2 - \alpha \cdot \frac{\partial \mathcal{L}}{\partial w_2} \qquad b \leftarrow b - \alpha \cdot \frac{\partial \mathcal{L}}{\partial b}$$

**This is it.**

You've just completed the hardest part of understanding how a neuron learns. And it wasn't magic. It was math. And you walked through every step of it.
Let me say it plainly: **most students in engineering schools don't know how to do this.** But you just did it. From scratch. Without skipping the hard parts.

All that's left now is to put this into code - to turn the math into a living neuron that learns from data.

### → Head over to the `practice_00` notebook `(Next Page)` to build and train your first intelligent neuron.

Let's watch it learn to separate toxic from non-toxic plants - all by itself.

**Launch `practice_00.ipynb` on Google Colab**

*Follow this part on Google Colab here.*
*Walk through the logic, line by line.*
*Clear pre-coded space, detailed commentary, and full freedom to experiment.*

# Here, We Code Our First Artificial Neuron

## Theory in Short

A neuron is simply a linear model with two learnable parameters: **W** and **b**. Our goal is to find the best values for these parameters so that the neuron makes good predictions.

To do that, we loop through a learning procedure based on four essential steps:

1. **Model Function**
   Defines how the neuron behaves - a linear transformation followed by an activation function (like sigmoid).

2. **Cost Function**
   Measures the error between predicted outputs and ground-truth labels $y$. It quantifies how well the neuron performs.

3. **Gradients Function**
   Computes the derivatives of the cost function with respect to the parameters - telling us in which direction to adjust them.

4. **Update Function**
   Applies the gradients to adjust $W$ and $b$ slightly, using a learning rate $\alpha$.

   This is the generic learning architecture behind **every neural network** - from a single neuron to GPT-4.

## Message for Non-Coders

Don't get lost in the code. You're not here to prove you're a Python wizard. You're here to understand the architecture - and how the pieces fit together.

Below, you'll see us importing:

- `numpy` (`np`) to handle arrays and vector operations
- `matplotlib` (`plt`) to plot data and learning curves
- `sklearn` to generate a toy dataset

Don't panic if you don't know what every line does. That's not the point. What matters is this:

**You don't need to know how every tool works under the hood.** You need to know what the tool does - and why you're using it.

That's enough to build a neuron. That's enough to build intelligence. So keep that mindset - and walk through the fire.

### Let's *funking* code.

**Let's build our first neuron**

We're about to code our very first artificial neuron. You'll see: once the structure is in place, everything else becomes modular.
This is the real beginning of your deep learning journey. **No magic, no shortcuts - just logic.**

```
# Install the necessary libraries (run only once)
# %pip install numpy matplotlib scikit-learn

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
```

### The Dataset - Let's Keep It Simple

Before doing anything in deep learning, we need data. Instead of scraping, cleaning, or searching for a dataset, we'll generate one. Why? Because we want full control to understand every step.

We'll simulate a binary classification problem:

- Class **0**: Non-toxic plants
- Class **1**: Toxic plants

And each plant is represented by two features:

- $x_1$: leaf length
- $x_2$: leaf width

```
# Generate a dataset of 100 plants with 2 features (length, width)
X, y = make_blobs(n_samples=100, n_features=2, centers=2, random_state=0)

# Reshape y to column vector shape (100, 1)
y = y.reshape((y.shape[0], 1))

# Display shapes to confirm structure
print('X shape:', X.shape)
print('y shape:', y.shape)
```

We now have:

- `X`: shape `(100, 2)` - 100 plants $\times$ 2 features
- `y`: shape `(100, 1)` - labels for each plant

### Visualizing the Dataset

```
# Plot the dataset
plt.figure(figsize=(6, 5))
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis', edgecolors='k')
plt.title("Toxic vs Non-Toxic Plants")
plt.xlabel("Leaf Length (x1)")
plt.ylabel("Leaf Width (x2)")
plt.grid(True)
plt.tight_layout()
plt.show()
```

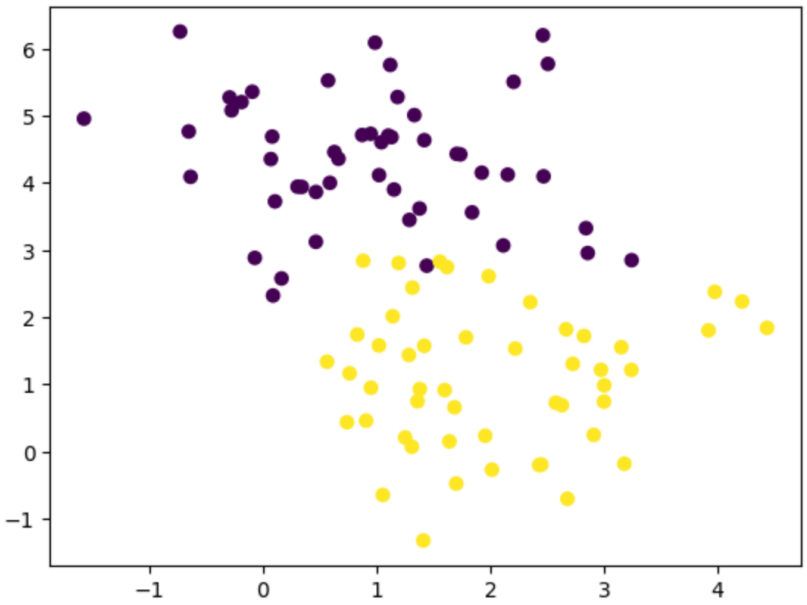



Figure 5: Visual separation of toxic (1) and non-toxic (0) plants

**Pro tip:** Always visualize your dataset before modeling. You want to understand how your data is structured and if the problem is linearly separable.

### Initialization - The Starting Line

**Step 1: Initialize the Model**

Every learning algorithm needs a starting point. Before the neuron can learn anything, we need to define what it starts with - that's where parameter initialization comes in.

We define a simple `initialisation` function to set up our model's parameters:

- $W$ is a vector of weights - one per feature (here: $x_1$, $x_2$).
- $b$ is the bias - a scalar added after the weighted sum.

Since our data points lie in 2D, we initialize:

$$W \in \mathbb{R}^{2\times 1}, \quad b \in \mathbb{R}$$

```python
def initialisation(X):
    W = np.random.randn(X.shape[1], 1)  # Random weight vector (2, 1)
    b = np.random.randn(1)              # Single bias term
    return W, b

# Apply initialization
W, b = initialisation(X)

# Display results
print(W.shape, b.shape)
print(W, b)
```

Suppose this gives us:

```
(2, 1) (1,)
[[-0.41675785]
 [-0.05626683]] [0.78522798]
```

**Premium tip:** Proper dimensions are everything. If your $W$ shape doesn't match your input features, your model won't even start. Always check it before moving forward.

We now have:

$$W = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}, \quad b \in \mathbb{R}$$

Let's move on to the neuron's forward computation.

## The Model - Neuron Behavior

**Step 2: Process and Activate**

This is the heart of the neuron. It receives input, transforms it linearly, and decides whether to activate - not with a brain, but with math.

Let's break it down:

- **Linear transformation:** $Z = X \cdot W + b$
- **Activation:** $A = \sigma(Z) = \frac{1}{1+e^{-Z}}$

The activation function introduces non-linearity. That's what makes the neuron smart - otherwise, it's just linear algebra.

```python
def model(X, W, b):
    Z = np.dot(X, W) + b              # Linear transformation
    A = 1 / (1 + np.exp(-Z))          # Sigmoid activation
    return A

A = model(X, W, b)
print(A.shape)
```

Suppose this returns:

```
(100, 1)
```

> **Why does this matter?** Because `A` must have the same shape as `y`, the ground truth. Only then can we compare the prediction with the actual label for each data point.

The model now outputs a probability for each input:

$$A = \text{sigmoid}(X \cdot W + b)$$

Each value in $A$ is the neuron's confidence that the corresponding plant is toxic.

Next, we'll need a way to evaluate whether these predictions are any good - and that's where the **cost function** comes in.

## Cost Function - Log-Loss

**Step 3: Measuring the Error**

How do we know if our neuron is doing well? We compare its prediction to the truth - and quantify the difference using a cost function.

In binary classification, the most common cost function is the **log-loss**, also called **binary cross-entropy**. It measures how far the predicted probability $A$ is from the actual class label $y$.

$$\text{Log-loss} = -\frac{1}{m}\sum_{i=1}^{m}(y_i \cdot \log(a_i) + (1 - y_i) \cdot \log(1 - a_i))$$

In Python:

```python
def log_loss(A, y):
    m = len(y)
    loss = -1/m * np.sum(y * np.log(A) + (1 - y) * np.log(1 - A))
    return loss

loss = log_loss(A, y)
print(loss)
```

> **What is this value?** A single real number that captures the model's *average prediction error*.

- If `loss` $\to 0$: the model predicts very well.
- If `loss` $\to 1$ (or higher): the model is confused and likely making poor predictions.

Our mission during training?

**Minimize the log-loss.**

Next, we'll need to calculate **how to change** our parameters to reduce that loss - and that's where **gradients** come in.

## Gradients - Learning How to Learn

**Step 4: Which Way to Go?**

To minimize the cost, the model needs to know: *Which direction should I go? And how fast?* That's the job of the **gradient**.

The gradient tells us how the cost changes with each parameter - in other words, how each weight and bias contributes to the prediction error.

We compute the gradients of the log-loss with respect to $W$ and $b$:

$$\frac{\partial \mathcal{L}}{\partial W}, \quad \frac{\partial \mathcal{L}}{\partial b}$$

In Python, it looks like this:

```python
def gradients(A, X, y):
    m = len(y)
    dZ = A - y                  # error term
    dW = 1/m * np.dot(X.T, dZ)
    db = 1/m * np.sum(dZ)
    return dW, db

dW, db = gradients(A, X, y)
print("dW = ", dW)
print("db = ", db)
```

- `dW` is a matrix of the same shape as $W$ - a gradient for each weight.
- `db` is a scalar - the gradient for the bias.

**These gradients point in the direction of maximum error increase.** So we'll go the *opposite* way - that's gradient descent.

In the next step, we'll use these gradients to **update the parameters** - and improve our model, step by step.

## Parameter Update - Learning by Doing

**Step 5: Adjust and Improve**

Now that we know *where* the error comes from (thanks to the gradients), let's move our parameters in the right direction - **toward less error**.

This is the heart of the **gradient descent** algorithm. We apply small updates to our weights and bias to reduce the cost, step by step.

**Update rules:**

$$W = W - \alpha \cdot \frac{\partial \mathcal{L}}{\partial W} \qquad b = b - \alpha \cdot \frac{\partial \mathcal{L}}{\partial b}$$

- $W$ is the weight vector
- $b$ is the bias term
- $\alpha$ is the **learning rate** - it controls the step size of each update

The learning rate is crucial:

- If it's too **small**, learning is slow
- If it's too **large**, the model might overshoot and never converge

```python
def update(W, b, dW, db, learning_rate=0.1):
    W = W - learning_rate * dW
    b = b - learning_rate * db
    return W, b

W, b = update(W, b, dW, db)
```

**Each update moves the model one step closer to the optimal parameters.** Train long enough, and the neuron will *learn* how to predict.

## Training Loop - Putting It All Together

**Step 6: Learn from Experience**

We now bring everything together: From initialization to prediction, from error measurement to learning. This is the full training pipeline of our artificial neuron.

**What we give the model:**

- `X` - the input data (leaf length and width)
- `y` - the true class labels (toxic or not)

**What the model handles:**

1. Initialization of parameters $W$, $b$
2. Forward pass (prediction)
3. Cost computation
4. Gradient calculation
5. Parameter update using gradients and learning rate

This whole process is repeated for multiple iterations. At each iteration, the neuron becomes a bit better.

```python
def artificial_neuron_test(X, y, learning_rate=0.1, num_iter=100):
    # Initialize parameters
    W, b = initialisation(X)

    # Keep track of the loss
    loss = []

    for i in range(num_iter):
        A = model(X, W, b)
        loss.append(log_loss(A, y))
        dW, db = gradients(A, X, y)
        W, b = update(W, b, dW, db, learning_rate)

    # Plot the learning curve
    plt.plot(loss)
    plt.title("Learning Curve")
    plt.xlabel("Iterations")
    plt.ylabel("Log Loss")
    plt.grid(True)
    plt.show()
```

Run the training:

```python
artificial_neuron_test(X, y)
```

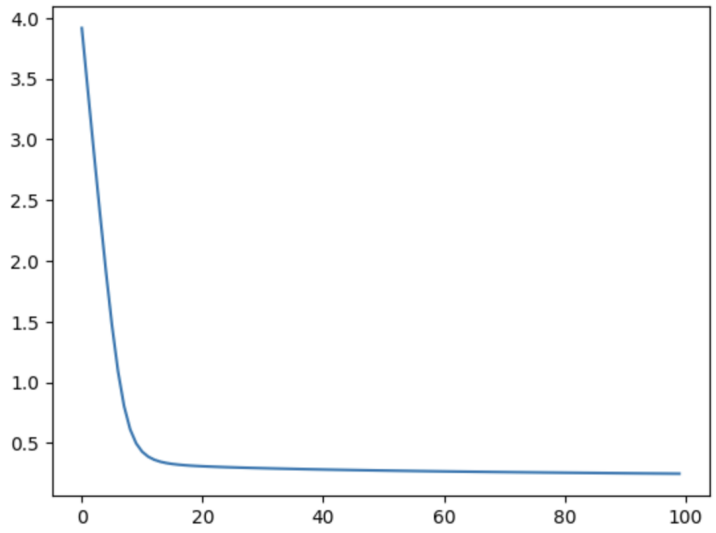



Figure 6: Learning Curve - Log Loss across iterations

**This is what learning looks like.** The model becomes less wrong over time.

We trained the model for 100 iterations, which means the model had 100 training sessions (or **epochs**) to improve.

## From Loss to Accuracy: A Subtle but Important Shift

Up until now, we've been using the **loss function** to evaluate our model - and that's essential. The loss tells us how well the model fits the data during training. It's a continuous value that reflects the quality of the prediction *probabilities*.

But there's a catch: **loss is not accuracy**.

What's the difference?

- The **loss** measures how far the predicted probabilities are from the correct labels. It's sensitive to confidence - predicting `0.51` when the correct class is `1` is penalized less than predicting `0.01`.
- The **accuracy** simply measures: *Did we get the class right? Yes or no?*

  We round the sigmoid output to `1` if it's $\geq 0.5$, and to `0` otherwise. Then we compare this prediction to the true label.

This is why we now define a new function: `predict()`.

It doesn't return probabilities - it returns the predicted class (`0` or `1`) for each input.

Once we have that, we can compute the model's **accuracy** by counting how many predictions match the actual labels.

This gives us a simple but powerful insight:

> *Loss guides the learning. Accuracy tells us if it worked.*

In our case, the question is: out of 100 plants, how many were classified correctly?

## So, How Do We Predict?

Our model already gives us the output $A$ - a probability between 0 and 1 - using the sigmoid function.

Now we turn that into a final decision:

- If $A \geq 0.5 \Rightarrow$ the plant is **toxic** (Class 1)
- If $A < 0.5 \Rightarrow$ the plant is **non-toxic** (Class 0)

This leads us to define the `predict()` function:

```
def predict(X, W, b):
    A = model(X, W, b)
    return A >= 0.5
```

This function transforms a **probability into a decision**.

The closer $A$ is to 0 or 1, the more confident the model is. The closer it is to 0.5, the less certain it becomes.

To compute the overall **accuracy**, we compare the predicted labels to the ground truth, and compute the proportion of correct predictions:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of examples}}$$

This final step closes the loop. You trained a neuron - now it answers.

```
from sklearn.metrics import accuracy_score

```

## Let's Test Our Trained Neuron on the Data

Now that training is complete, we want to see how well our neuron performs - in simple terms: *How many plants did it correctly classify as toxic or non-toxic?*

```
def artificial_neuron(X, y, learning_rate=0.1, num_iter=100):
    # Initialize parameters
    W, b = initialisation(X)

    # Track loss for each iteration
    loss = []

    for i in range(num_iter):
        A = model(X, W, b)
        loss.append(log_loss(A, y))
        dW, db = gradients(A, X, y)
        W, b = update(dW, db, W, b, learning_rate)

    # Training finished - now predict using trained parameters
    y_pred = predict(X, W, b)
    print('Accuracy score:', accuracy_score(y, y_pred))

    # Plotting the loss curve
    plt.plot(loss)
    plt.title("Training Loss over Iterations")
    plt.xlabel("Iterations")
    plt.ylabel("Log-loss")
    plt.grid(True)
    plt.show()

    print("Final trained parameters:")
    print("W =", W)
    print("b =", b)

    return W, b

```

```
W, b = artificial_neuron(X, y)

```

You might obtain an accuracy score like `0.92` or `0.86` - meaning:

> *The neuron correctly predicted whether a plant was toxic or not in 92% (or 86%) of cases.*

This prediction was made using only the leaf's width and length - a simple two-dimensional input.

You also recover the final, optimized parameters $W$ and $b$: the exact settings your neuron learned through gradient descent, representing the best configuration found over 100 training iterations.

> And yes - if the loss curve steadily decreases and plateaus, you've successfully trained your first artificial neuron.

## Prediction of a new data

What if you come with a new plant? Let's see.

```
new_plant = np.array([2, 1])
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis')
plt.scatter(new_plant[0], new_plant[1], c='r')
plt.show()
```

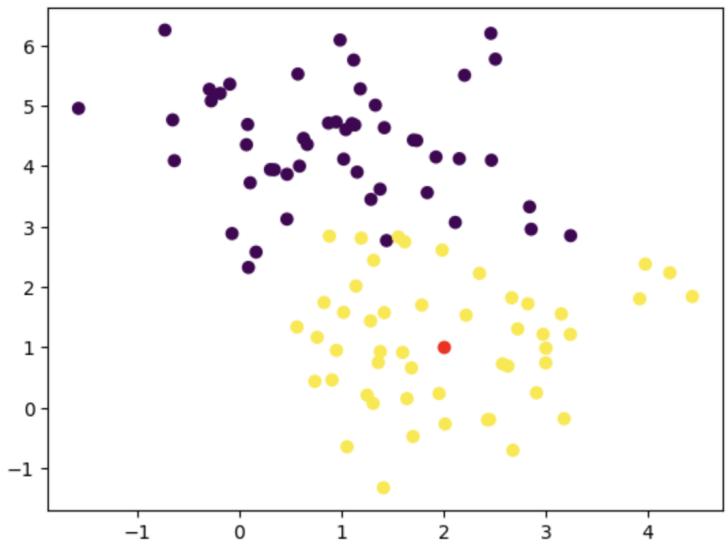



Figure 7: New plant (red) among known data points

The red point represents a new plant whose leaf length and width have just been measured. We place it on the same feature space as our training data.

Now, the question is simple: **What would the model predict for this new plant?**

Let's compute its probability of being toxic, and make a prediction based on the output of our trained neuron.

```
def predict_plant(X, W, b):
    A = model(X, W, b)
    print(A)
    return A >= 0.5

predict_plant(new_plant, W, b)
```

You obtain `True` because the predicted probability is **0.93**, which is greater than the decision threshold of 0.5. This means the neuron confidently classifies the plant as **toxic**.

According to the model, this new plant belongs to class 1 - and we can trust this result with **93% confidence**.

This illustrates a crucial concept: the neuron's prediction is not just a binary output - it's a probability. A probability that reflects the model's **degree of certainty**.

> Some points are classified with high confidence (probabilities close to 0 or 1). Others fall near the threshold of 0.5 - where uncertainty is highest.

If we visualize all points for which the model assigns a probability of exactly 0.5, we obtain a transition line: the **limit where the neuron hesitates**.

This is called the **decision boundary** - the frontier between one class and the other. A boundary that the model learns through training, and refines as it gains experience.

> Just beyond this invisible line, the model is able to decide. Right on it - it hesitates.

## Let's Plot the Decision Boundary

From the math behind the activation function, saying that $\boldsymbol{a = 0.5}$ is equivalent to saying that $\boldsymbol{z = 0}$.

Let's draw the corresponding line - the decision boundary:

```
new_plant = np.array([2, 1])

x1 = np.linspace(-1, 4, 100)
x2 = (-W[0] * x1 - b) / W[1]

plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis')
plt.scatter(new_plant[0], new_plant[1], c='r')
plt.plot(x1, x2, c='orange')
plt.show()
```

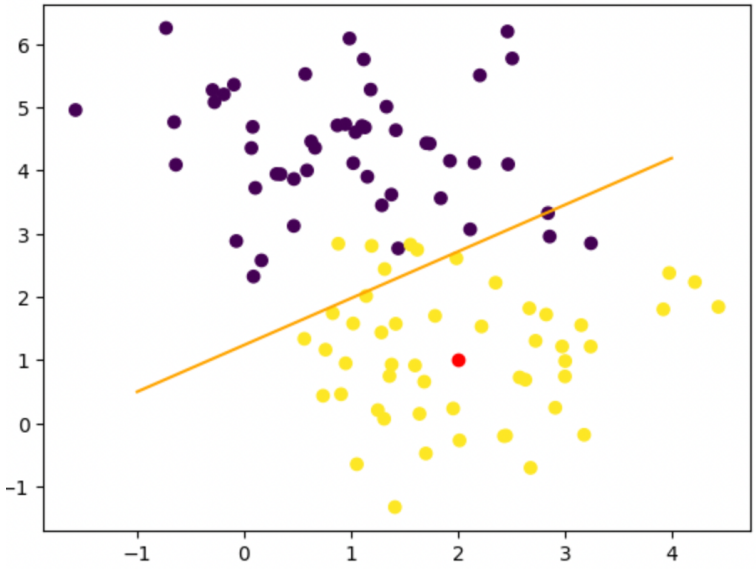



Figure 8: The decision boundary separates toxic and non-toxic classes.

**What do we see?**

- First, the **orange line** represents the decision boundary - learned by the neuron as a *linear model*.
- This line separates the two classes: **toxic** vs **non-toxic**.

- The **red point** (our new plant) lies clearly on one side - far from the boundary.
- This means the neuron had **high confidence** when classifying it as toxic.
- On the other hand, points **closer to the boundary** are more ambiguous.

Why? Because when $z \approx 0$, the activation becomes $a \approx 0.5$ - and the neuron is **unsure** about its decision.

**At the decision boundary, it's like flipping a coin.** The model is maximally uncertain - and errors are more likely.

Even if we train the model for a long time (many epochs), this type of neuron - a **linear classifier** - has **inherent limitations**.

**It can only learn linear boundaries.**

So while it performs well in many cases, it may struggle when the true boundary between classes is curved, complex, or nonlinear.

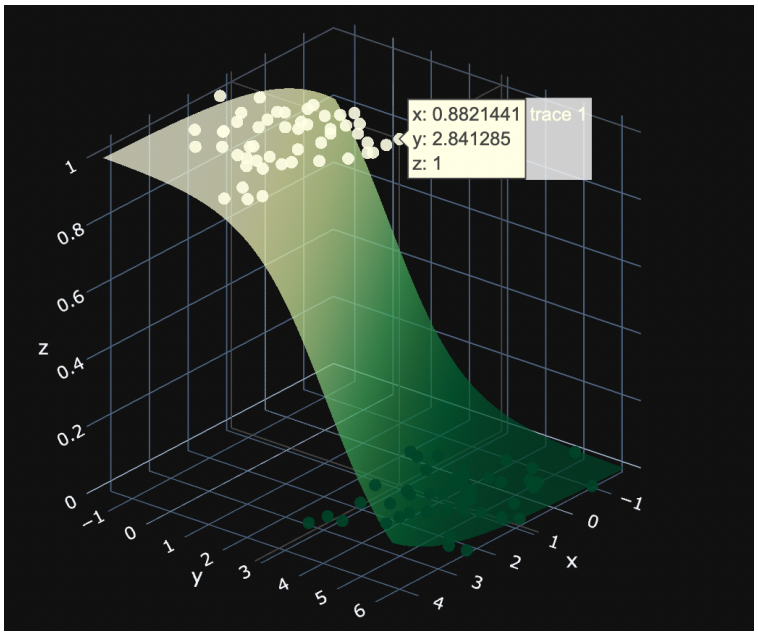



Figure 9: Points near the boundary are the most ambiguous

This is another view of the decision-making process in the 2D space, using the **sigmoid function** as our activation.

As we can see from the plot:

- The sigmoid outputs values between 0 and 1.
- If the output is close to **1** $\rightarrow$ the model predicts the plant is **toxic**.
- If the output is close to **0** $\rightarrow$ the model predicts the plant is **non-toxic**.

The **center of the sigmoid** - where outputs are near 0.5 - is where predictions are the most difficult. Many of the data points near the decision boundary lie in this ambiguous region, and some of them are **likely misclassified**.

**This is why the sigmoid is so useful in binary classification:** It transforms linear outputs into smooth probabilities between 0 and 1.

## So congrats - you just understood a binary classification model.

## Going Beyond 2D: Higher-Dimensional Data

In real life, we rarely work with just two input variables. Datasets can include **10**, **100**, or even **10,000 features**.

In such cases, it becomes impossible to visualize the data in simple 2D plots like the ones we've used so far.

**So how do we evaluate the model?**

We rely on metrics such as the **learning curve** - which tracks how the loss evolves over time - to assess the model's performance.

That's why we are going to use this neuron to classify images of cats and dogs. Let's go for the `practice_01.ipynb` notebook.

practice_01.ipynb

**Launch `practice_01.ipynb` on Google Colab**

*Follow this part on Google Colab here.*
*Code and Test the neuron on real images.*
*See where it struggles, and experiment this pre-coded space as you wish.*

## Practice: Cat vs Dog

Let's put our neuron to the test - can it classify actual images?

We'll use a small dataset of pictures showing either **cats** or **dogs**, and see if the neuron can learn to tell them apart.

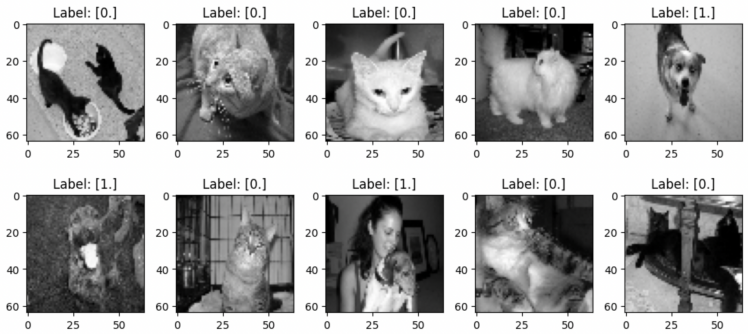



Figure 10: Some photos of the dataset

To do this, we'll use two separate datasets:

- One for **training** the neuron.
- One for **testing** the neuron's ability to generalize.

## Why do we use two datasets?

If we trained and tested on the same data, the neuron might just *memorize* the answers, without truly learning how to generalize.

By testing on **unseen images**, we verify whether the neuron actually **understands the task**.

It's just like preparing for an exam: practicing with exercises is helpful, but the real test is solving **new problems**.

Let's load the neuron code and the dataset:

```python
from practice_00 import artificial_neuron, initialisation, model, gradients, \
                        update, predict, accuracy_score
from sklearn.metrics import accuracy_score
```

```python
X_train, y_train, X_test, y_test = load_data()
```

```python
print("X_train shape: ", X_train.shape)
print("y_train shape: ", y_train.shape)
print(np.unique(y_train, return_counts=True))
```

**Training Data Overview**

- `X_train shape: (1000, 64, 64)`
- `y_train shape: (1000, 1)`
- Labels: `array([0., 1.])`, with `array([500, 500])`

## Understanding the Training Dataset

We have:

- **1,000 images** in total - each of size **64 × 64 pixels**.
- Each image is labeled as either:
  - `0` for a **cat**,
  - `1` for a **dog**.
- The dataset is **perfectly balanced** with 500 cats and 500 dogs.

Let's check the test dataset as well:

```python
print("X_test shape: ", X_test.shape)
print("y_test shape: ", y_test.shape)
print(np.unique(y_test, return_counts=True))
```

**Test Data Overview**

- `X_test shape: (200, 64, 64)` - 200 unseen test images.
- `y_test shape: (200, 1)` - corresponding labels.
- Labels: `array([0., 1.])`, with `array([100, 100])` - balanced test set.

## How Do We Feed a Photo into Our Artificial Neuron Model?

At this point, you might be wondering:

*"But how do we actually give a photo to our neuron?"*

**The answer is surprisingly simple.** We treat each pixel in the image as a separate input variable.

Since each photo is **64 × 64 pixels**, we have:

$$64 \times 64 = \mathbf{4096} \text{ input variables}$$

We scan the image row by row - from left to right, top to bottom - and flatten the pixels into a single long vector:

$$\boldsymbol{X = (x_1, x_2, \ldots, x_{4096})}$$

Each value $x_i$ represents the intensity of one pixel.

**A photo becomes just another numerical input.** From there, the process is identical to what we did before: we apply a linear model to $X$ and compute a prediction.

If the output is close to 1 ⇒ it's a **dog**. If it's close to 0 ⇒ it's a **cat**.

**The neuron doesn't see images - it sees numbers.**

## So What Does the Full Process Look Like?

Let's step back for a second and look at the big picture: **How do we go from a raw photo to a final prediction?**

Here's the roadmap - four essential steps:

1. **Normalize the input.** Each pixel is stored on 8 bits (0 to 255). To simplify learning, we rescale all values between 0 and 1:
$$x_i \leftarrow \frac{x_i}{255}$$
2. **Flatten the image.** The photo (64 × 64 pixels) becomes a 4096-dimensional input vector:
$$X = (x_1, x_2, \ldots, x_{4096})$$
3. **Train the model.** Feed labeled examples into the neuron, let it make predictions, compare them to the true labels, and adjust the weights accordingly.
4. **Evaluate generalization.** Use a separate test set - unseen during training - to check if the neuron has truly learned to distinguish cats from dogs.

**That's your pipeline.** From raw pixels to meaningful prediction - in just four simple steps.

## Steps 1 & 2 - Let's Transform and Normalize at the Same Time

Before feeding our images into the neuron, we need to prepare them properly. That means doing two things at once:

- **Flattening** the images - turning each $64 \times 64$ photo into a single vector of 4096 values.
- **Normalizing** the pixel values - scaling them from $[0, 255]$ to the range $[0, 1]$.

Why divide by 255? Because each pixel is stored on 8 bits, meaning it can take $2^8 = 256$ different values - from 0 to 255.

**Why normalize the data in the first place?**

Imagine you're in a band, doing a sound check before a concert at La Cigale, Paris. Each instrument - drums, guitar, vocals - plays its part. But if the drums are way louder than everything else, they overpower the mix.
In data science, it's the same. If one feature has much larger values than the others, the model might "hear" it more - but that doesn't make it more important.
Normalization is like adjusting the volume levels: it ensures that every feature contributes fairly - so the model listens to the whole band, not just the loudest player.

**In one sentence: we normalize to be fair.**

Let's now see how we can normalize and flatten our data at the same time:

```
X_reshaped = []
for image in X_train:
    image_flat = []
    for row in image:
        for pixel in row:
            image_flat.append(pixel / 255)
    X_reshaped.append(image_flat)

X_reshaped = np.array(X_reshaped)
print(X_reshaped.shape)
print("Pixel 35 of image 465:", X_reshaped[465][35])

```

Output:

```
X_reshaped.shape: (1000, 4096)
Pixel 35 of image 465: 0.03868
```

We observe two important things:

1. Each pixel has been properly **normalized**.
2. We now have a `(1000, 4096)` input matrix, where:

- Each **row** is a flattened image (1000 in total),
- Each **column** corresponds to a specific pixel index across all images.

But in practice, we don't need to write these loops ourselves. Python and NumPy make it easy to reshape the data:

```
X_train_reshape = X_train.reshape(X_train.shape[0], X_train.shape[1] * X_train.
    shape[2]) / 255
```

```
X_train_reshape shape: (1000, 4096)
Pixel 35 of image 465: 0.03868
```

But maybe the pixels are coded on 16 bits instead of 8. To be sure, we have to divide by the maximum value of the pixel found in the dataset. This is why we use the `X_train.max()` function to find the maximum value of the pixels in the dataset.

And for those already comfortable with Python and NumPy, here's the fully vectorized version - short, elegant, and efficient:

```
X_train_reshape = X_train.reshape(X_train.shape[0], -1) / X_train.max()
X_test_reshape = X_test.reshape(X_test.shape[0], -1) / X_test.max()
```

**At This Point...**

Your training images are now ready - each one is a clean, normalized vector of 4096 values. Your neuron can now start learning from the data.

## 3. Time to Train the Model on the Training Data

Let's try training our neuron:

```
W, b = artificial_neuron(X_train_reshape, y_train, learning_rate=0.1, num_iter
    =100)
```

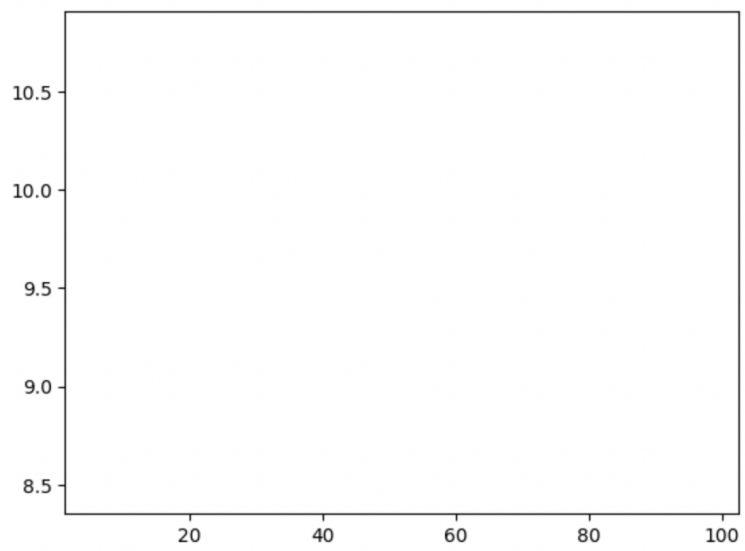



Figure 11: Nothing's happening ??

But... we hit an error.

*"RuntimeWarning: divide by zero encountered in log"*

Let's break it down:

- The error comes from the `log_loss` function. - Inside it, we compute a logarithm that depends on the predictions `A`. - `A` comes from the sigmoid activation, which uses the exponential function.

> **The exponential grows fast - really fast.** Even modest values like $z = 20$ make $\exp(-z)$ close to 0 which leads to values of $A$ near 1 or 0 - and $\log(0)$ is undefined.

**Fix**: Add a small constant $\varepsilon$ to prevent $\log(0)$. This is a common practice in deep learning to avoid numerical instability. Let's add it to our `log_loss` function:

```
# the problem is in log_loss
def log_loss(A, y):
    m = len(y)
    epsilon = 1e-15
    loss = -1/m * np.sum(y * np.log(A + epsilon)
                    + (1 - y) * np.log(1 - A + epsilon))
    return loss
```

Now let's define the updated neuron training function with the new `log_loss` function:

```
def new_artificial_neuron(X, y, learning_rate=0.1, num_iter=100):
    W, b = initialisation(X)
    loss = []
    for i in range(num_iter):
        A = model(X, W, b)
        loss.append(log_loss(A, y))
        dW, db = gradients(A, X, y)
        W, b = update(dW, db, W, b, learning_rate)
```

```
    y_pred = predict(X, W, b)
    print("Accuracy score:", accuracy_score(y, y_pred))
    plt.plot(loss)
    plt.xlabel("Iterations")
    plt.ylabel("Log Loss")
    plt.title("Training Loss Curve")
    plt.show()
```

Now let's try training again:

```
new_artificial_neuron(X_train_reshape, y_train)
```

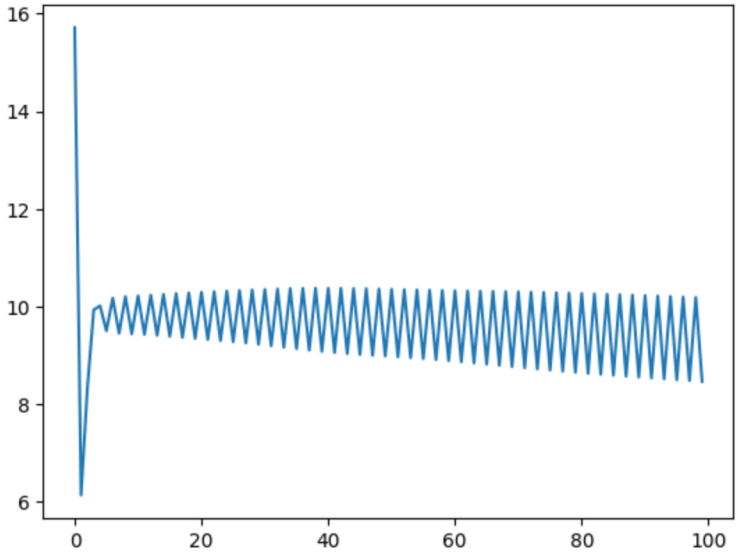



Figure 12: That's better.. I guess ?

## It Works - But the Behavior Is Unstable

The model now runs without crashing. But the learning curve is shaky, and the accuracy may vary wildly.

**Why?**
Because of the **hyperparameters**. Just like normalization, they play a crucial role in how well the model learns.

We'll now explore how learning rate, number of iterations, and weight initialization affect performance.

## Hyperparameters and Diagnosis

Hyperparameters are called that way because they're **hyper important** - they are the only levers we can use to **control how the neuron learns**, once it's built.

Here are two key examples:

- **Learning rate** - controls how big each update is. Too high? The model may overshoot. Too low? It might never converge.
- **Number of iterations (a.k.a. epochs)** - how many times we let the neuron learn from the data. Too few? It doesn't learn. Too many? It might overfit or get stuck.

> **Hyperparameters = the teacher's schedule.** They decide how fast the neuron moves, and how long the learning continues.

We'll now test how different values for these hyperparameters affect the training behavior - and learn how to diagnose when something goes wrong.

## Learning Rate

The **learning rate** controls the influence of the gradient on the weight updates.

If it's too high, the model might **overshoot** the optimal point - bouncing back and forth, never settling. If it's too low, the model learns **very slowly** - or not at all.

Let's reduce the learning rate from 0.1 to 0.01, and test two training durations:

```
new_artificial_neuron(X_train_reshape, y_train, learning_rate=0.01, num_iter=100)
new_artificial_neuron(X_train_reshape, y_train, learning_rate=0.01, num_iter=1000)
```

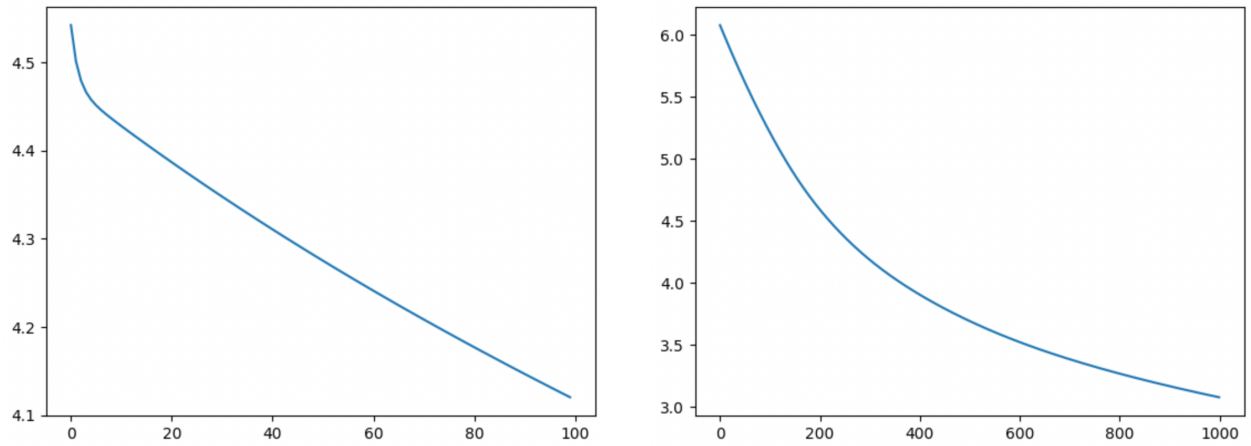



Figure 13: Learning curve with 100 and 1000 iterations (learning rate = 0.01).

That's much smoother. To better understand what's going on, let's modify our neuron to track the loss and accuracy over time.

```
def new_artificial_neuron(X, y, learning_rate, num_iter):
    W, b = initialisation(X)

```

```
    # Initialize lists to store loss and accuracy
    loss = []
    acc = []

    for i in range(num_iter):
        A = model(X, W, b)

        if i % 50 == 0: # to reduce the number of points
            loss.append(log_loss(A, y))
            y_pred = predict(X, W, b)
            acc.append(accuracy_score(y, y_pred))

        dW, db = gradients(A, X, y)
        W, b = update(dW, db, W, b, learning_rate)

    plt.figure(figsize=(12, 4))
    plt.subplot(1, 2, 1)
    plt.plot(loss)
    plt.title("Loss")

    plt.subplot(1, 2, 2)
    plt.plot(acc)
    plt.title("Accuracy")
    plt.show()
```

```
new_artificial_neuron(X_train_reshape, y_train, learning_rate=0.01, num_iter=1000)
```

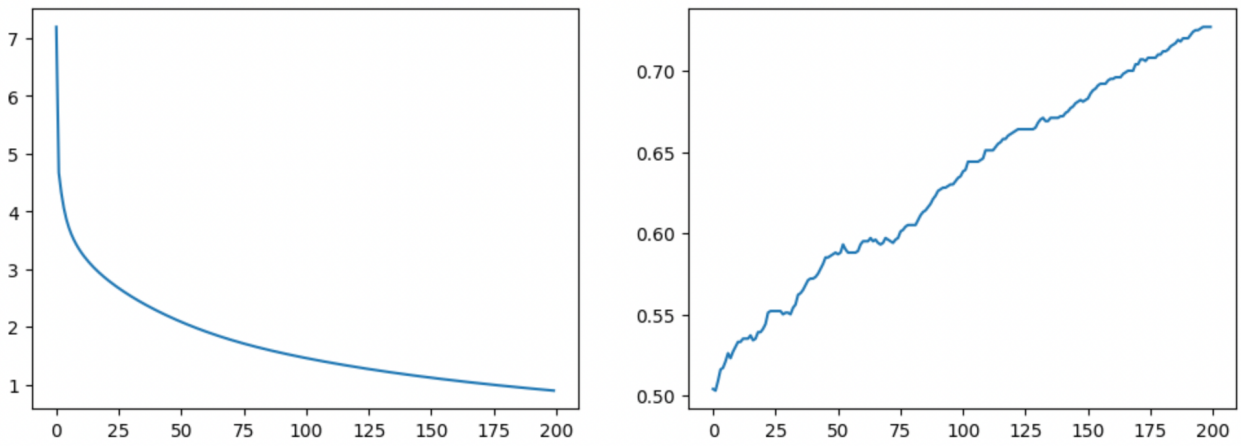



Figure 14: Loss and accuracy over time (learning rate = 0.01, epoch = 1000).

We observe something interesting:

- The **loss** curve decreases - and then flattens as the model reaches a minimum.
- Meanwhile, the **accuracy** continues to improve - even after the loss has plateaued.

## Is the Model Overfitting?

At this point, one of the most likely explanations is that the model is entering into a state of **overfitting**.

**Overfitting** occurs when a model performs extremely well on the training data - but fails to generalize to new, unseen examples.

It's as if the neuron has studied only the *answers* to the exercises, without learning the *method* behind them. It memorizes the training set - but doesn't understand the broader logic.

> Just like a student who aces the practice questions, but struggles on the exam - because the questions are phrased differently.

So how can we tell if this is what's happening?

We need to test the model on data it hasn't seen - for instance, the `test set` - and compare its performance.

Let's modify our training function to track both training and test metrics:

```
def artificial_neuron(X_train, y_train, X_test, y_test, learning_rate=0.1, n_iter
    =100):
    # initialization
    W, b = initialisation(X_train)

    train_loss = []
    train_acc = []
    test_loss = []
    test_acc = []

    for i in tqdm(range(n_iter)):
        A = model(X_train, W, b)

        if i % 10 == 0:
            # Train set metrics
            train_loss.append(log_loss(A, y_train))
            y_pred = predict(X_train, W, b)
            train_acc.append(accuracy_score(y_train, y_pred))

            # Test set metrics
            A_test = model(X_test, W, b)
            test_loss.append(log_loss(A_test, y_test))
            y_pred = predict(X_test, W, b)
            test_acc.append(accuracy_score(y_test, y_pred))

        # gradient descent
        dW, db = gradients(A, X_train, y_train)
        W, b = update(dW, db, W, b, learning_rate)

    plt.figure(figsize=(12, 4))
    plt.subplot(1, 2, 1)
    plt.plot(train_loss, label='train loss')
    plt.plot(test_loss, label='test loss')
    plt.title("Loss over time")
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(train_acc, label='train acc')
    plt.plot(test_acc, label='test acc')
    plt.title("Accuracy over time")
    plt.legend()
```

```
41
42      plt.show()
43
44      return (W, b)
```

Now let's launch the training with both datasets:

```
1 W, b = artificial_neuron(X_train_reshape, y_train, X_test_reshape, y_test,
      learning_rate=0.01, n_iter=10000)
```

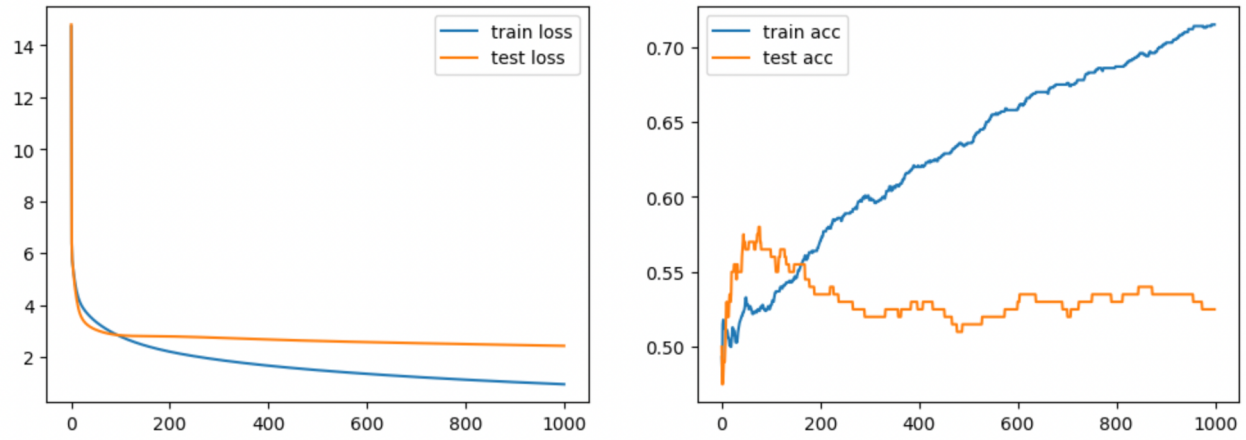



Figure 15: Really interesting...

## Detecting Overfitting

The plots reveal a clear divergence between the training and test curves.

- The **training loss** decreases steadily, and the accuracy climbs toward 100%.
- But the **test loss** stagnates - or even rises - and the test accuracy lags behind.

This is a textbook case of **overfitting**. The model is becoming an expert on the training data - but it fails to generalize its knowledge to new, unseen examples.

> **Warning sign:** If the gap between training and test performance keeps growing, you're likely overfitting.

We'll now look at strategies to address this - such as reducing model complexity, using regularization, or training with more diverse data.

## How Can We Fix This Problem?

Overfitting is a common challenge in machine learning - but fortunately, we have several tools to fight it.

Here are a few strategies:

1. **Give the model more data.** By exposing the neuron to a greater variety of examples, it learns to generalize better.
2. **Simplify the input.** Reducing the number of input features (dimensionality reduction) can help the model focus on the most relevant information.
3. **Use regularization.** Techniques like **L1** and **L2** penalties discourage overly complex models by gently pushing weights toward zero.

But in our case, none of these will dramatically improve the performance. Why?

Because our model is just **too simple**.

Until now, we've only been using a **single neuron**. That's a linear model - it can only draw straight lines.

So if the data is not linearly separable - if the true boundary is curved, complex, nonlinear - then a single neuron just isn't enough.

## Improving the Model

Okay - so we've pushed our little neuron as far as it can go.

It learned to classify, it learned to adjust itself, and it gave us decent results. But now, it's hitting a wall.

And we know why: it's just a **linear model**. One straight line. One decision boundary. That's not enough to solve complex problems like image recognition or language understanding.

So what do we do?

We scale up.

**We add more neurons. We add more layers.** We move from a single-cell brain to a full neural network.

This is where it gets serious - and way more powerful.

We're about to build our first real **Artificial Neural Network (ANN)**.

**Let's move from a neuron... to a brain.**

And yes - that means entering the world of equations, matrices, and weight updates.

But don't worry. You only need to understand it once.

Then you'll be free to create your own architectures.

# Theory of Neural Networks

Hi there — you just made the decision to understand the theory of neural networks. *It's gonna be tough, but trust me — you'll be amazed, and more importantly, true to yourself.*

**Final Message of This Notebook**

**I know what you're thinking:**

> *"Those equations are ugly as hell."*

But let me say two things:

1. I don't care — we're here to learn the real stuff. This isn't a fashion show.
2. You only need to go through this **once**.

Remember — a data scientist is not a mathematician. They rarely compute things by hand. Instead, they rely on libraries to do the heavy lifting. So, your job is simple: understand it deeply, once and for all. After that, you're good.

**And that's the beauty of it:** very few people truly understand what's going on behind the scenes. But those who do? They're the ones capable of designing their own models and architectures.

**Let's begin**

No more talking — let's dive into the theory of neural networks.

We've already encountered the concept of a neuron through the perceptron model.

- It takes multiple inputs,
- Applies weights,
- Produces an output using a threshold activation function.

This is the foundation. And yes, the perceptron can classify data into two categories — but it quickly reaches its limits. **It's not powerful enough** for real-world complexity.

## Machine Learning Approach

To overcome this, traditional **machine learning** applies a clever workaround. We enrich the input space with manually engineered features: things like $x_1^2$, $x_2^2$, $x_1 x_2$, and so on.

This transforms a linear model into a more flexible one — usually polynomial. The process is called *feature engineering*.

**Feature Engineering**

The art of creating new features from existing ones — based on domain knowledge, intuition, and a bit of trial-and-error.

It works — but it's manual, tedious, and not scalable to complex data.

## Deep Learning Approach

Deep learning asks a new question:

**What if... we connected neurons together?**

Instead of manually building features, we let the network *learn them itself.*

How?

We stack layers of neurons — one after the other — forming what we call a **deep neural network**. Each layer learns to extract more abstract, more meaningful representations from the previous one.

*This is the essence of deep learning:*

**Key Idea**

Let the model automatically learn the best features by stacking multiple layers of neurons.

Each transformation brings us closer to understanding the data. Each layer does a little more, building up to a powerful representation.

In the end, the model becomes capable of learning incredibly complex functions — **and making remarkably accurate predictions.**

Sounds intense?

**Good.** That's how you know it's worth it.

We're going to build a neural network from scratch — using math and matrix operations.

## Let's Go

We're not here to waste time. Let's take the neuron we previously built — and simply **duplicate it**.

We now have two neurons: $N_1$ and $N_2$, placed side by side. And just like that, we've built our first:

## Layer

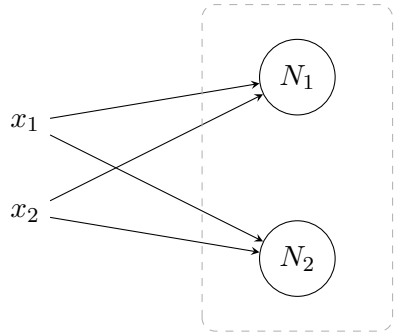



Figure 16: A layer groups multiple neurons — each with its own weights and bias — but sharing the same input.

## What Is a Layer?

A **layer** is simply a collection of neurons that operate *in parallel*. They all receive the same input vector, but each computes something slightly different.

- Each neuron has its own weight vector and bias.
- Each neuron produces one output.

So here, with $N_1$ and $N_2$, we'll get two outputs: $z_1$ and $z_2$.

## Let's Zoom In — Neuron by Neuron

**Neuron** $N_1$ takes the input $(x_1, x_2)$ and computes:

$$z_1 = W_{11} \cdot x_1 + W_{12} \cdot x_2 + b_1$$

Where:

- $W_{11}$ and $W_{12}$ are the weights of $N_1$,
- $b_1$ is its bias.

**Neuron** $N_2$ does the exact same thing — but with different weights and bias:

$$z_2 = W_{21} \cdot x_1 + W_{22} \cdot x_2 + b_2$$

So even though they process the same input, the result is different — because their parameters are different.

This is the power of layers: many neurons, one input, multiple outputs.

### The Output Equation — Matrix Form

Now let's write all of this in a single, elegant expression:

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Or, more compactly:

$$Z = Wx + b$$

**Notation Guide**

- $x$ is the input vector (shared by all neurons),
- $W$ is the weight matrix — each row is a neuron,
- $b$ is the bias vector,
- $Z$ is the output vector of the layer.

**Key Insight**

The matrix equation $Z = Wx + b$ allows us to compute the output of the whole layer in one step — it's the vectorized form of everything we just did manually.

## Activation Function

We're almost there. Now that our neurons have computed their raw outputs $z_1$ and $z_2$, we apply a final transformation:

**The activation function.**

Why? Because we want to introduce **non-linearity** into our model — so it can learn complex patterns, not just straight lines.

**Sigmoid Activation Function**

The sigmoid is a classic choice:

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

It maps any real number to a value between 0 and 1. Perfect when we want to interpret outputs as probabilities.

We apply the sigmoid to each neuron's output individually:

$$A = a(Z) \quad \Rightarrow \quad \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} a(z_1) \\ a(z_2) \end{bmatrix} = \begin{bmatrix} \frac{1}{1+e^{-z_1}} \\ \frac{1}{1+e^{-z_2}} \end{bmatrix}$$

Here, $a_1$ and $a_2$ are the **activated outputs** of neurons $N_1$ and $N_2$.

**Key Insight**

Each neuron computes its own linear combination: $z = Wx + b$ Then applies an activation: $a = \sigma(z)$ The result: nonlinear, learnable, expressive outputs.

## Visual Summary

Here is what we've just built — a layer of two neurons with activation functions:

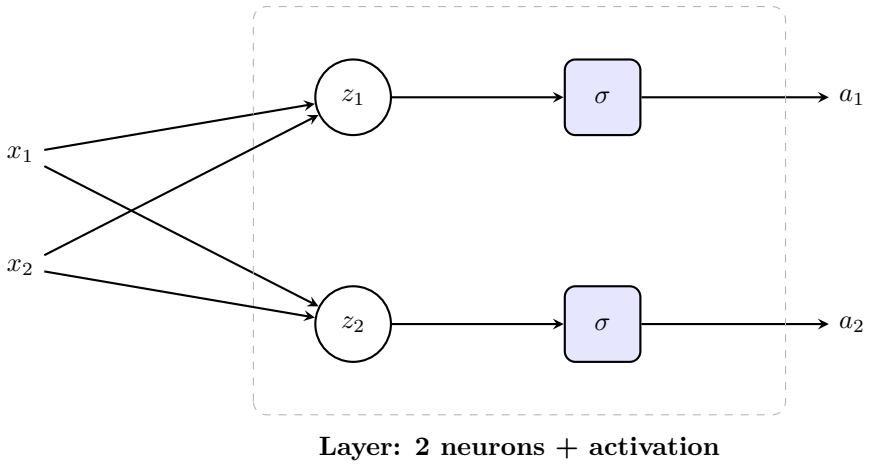



Figure 17: Each neuron computes $z = Wx + b$, then applies $\sigma(z)$ to produce its final output $a$.

## Very Important

**The outputs of the activation function are the true outputs of the layer.**

In our case, we now have two activated outputs:

$$\boxed{a_1 \quad \text{and} \quad a_2}$$

These form the output vector of the first layer.

## Adding Neurons in the Layer

Let's say we want more power. What's stopping us from adding more neurons to the layer?

Answer:

**Nothing.** We just add more rows to the weight matrix, and more entries in the bias vector.

Here's what that looks like:

$$W = \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \\ W_{31} & W_{32} \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Same input $x$, same formula:

$$Z = Wx + b$$

Now, the output becomes:

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \\ W_{31} & W_{32} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Each new row corresponds to a new neuron, with its own weights and bias. The logic remains exactly the same.

**We can add 3, 10, 100 neurons — as many as we want — and the equation stays the same.**

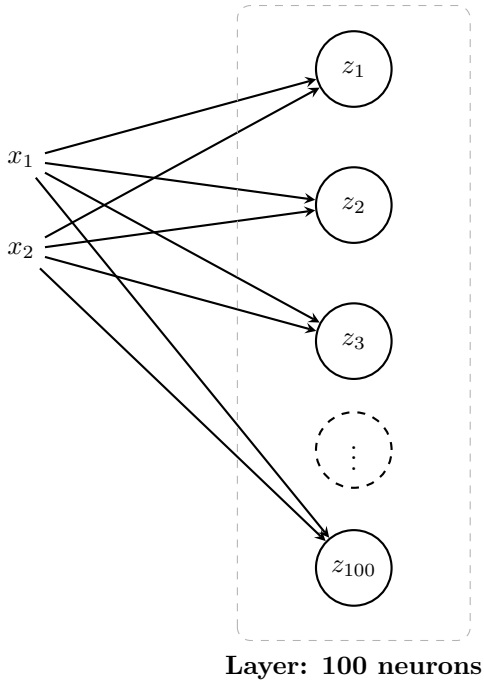



Figure 18: A layer with 100 neurons — each receiving the same input, but with its own weights and bias.

## A Layer is a Collection of Neurons

Let's take a step back and reframe what a layer really is.

A **layer** is just a collection of neurons working in parallel. It takes an input, and produces multiple outputs — one from each neuron.

**Analogy**

You can think of a layer as one giant neuron — but made of many smaller neurons, each with its own weights and bias.

For now, we're working with two neurons in our first layer.

We now introduce the notation for layers explicitly. The output of the first layer — after applying the activation — is written as:

$$A = A_1 = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = a(Z_1) = \begin{bmatrix} a(z_1) \\ a(z_2) \end{bmatrix}$$

Where:

- $Z_1$ is the pre-activation output of the first layer (before sigmoid),
- $A_1$ is the activated output of the first layer,
- $a_1$ and $a_2$ are the outputs of neurons $N_1$ and $N_2$ after activation.

## Adding a Second Layer

Now, something exciting happens.

We're no longer going to feed the raw input $x$ to the layer — we're going to feed it the output of the previous layer: $A_1$.

**Deep Architecture**

- The first layer takes the original input $x$ and produces $A_1$.
- The second layer takes $A_1$ as input and produces $A_2$.

Let's keep it simple for now. **We'll add just one neuron in the second layer.**
This neuron will:

- Take $a_1$ and $a_2$ as inputs,
- Apply its own weights and bias,
- Produce a pre-activation $z_3$,
- Then apply the activation to get $a_3$.

$$z_3 = W_{31} \cdot a_1 + W_{32} \cdot a_2 + b_3 \quad \Rightarrow \quad a_3 = \sigma(z_3)$$

So now we have:

$$A_2 = \begin{bmatrix} a_3 \end{bmatrix}$$

**Key Insight**

Each new layer takes the output of the previous layer as input. This stacking is what gives neural networks their depth — and power.

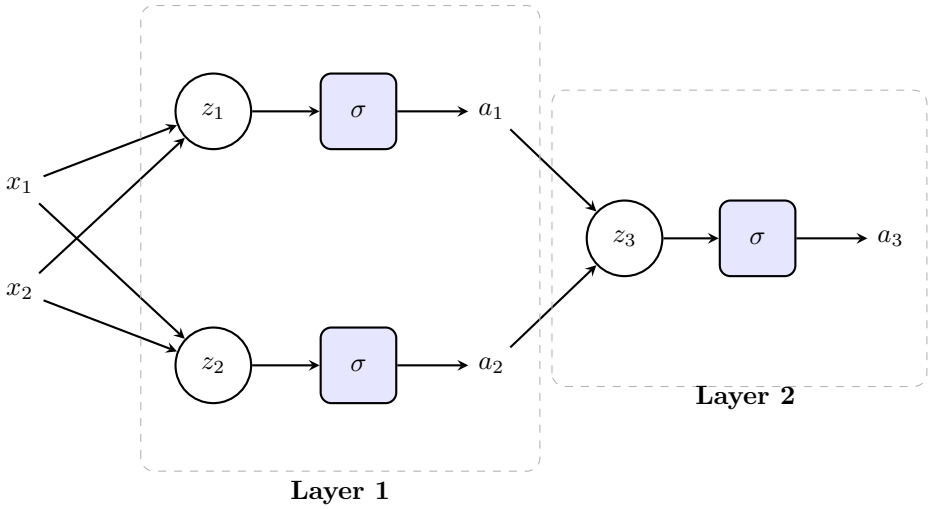



Figure 19: A two-layer neural network: the output of Layer 1 becomes the input to Layer 2.

## Second Layer Computation

We now know that the output of the first layer is:

$$A_1 = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$$

Let's feed this into the second layer. This time, we'll work with a single neuron in the second layer.

**Layer 2 Computation**

The second layer computes:

$$Z_2 = W_2 A_1 + b_2$$

Where:

- $W_2$ is the weight matrix of the second layer,
- $A_1$ is the input (from the first layer),
- $b_2$ is the bias of the second layer.

Then we apply the activation function:

$$A_2 = a(Z_2)$$

Let's look at a concrete example. Suppose we have:

$$W_2 = \begin{bmatrix} w_{21} & w_{22} \end{bmatrix}, \quad b_2 = \begin{bmatrix} b_3 \end{bmatrix}$$

Then:

$$Z_2 = \begin{bmatrix} z_3 \end{bmatrix} = W_2 A_1 + b_2 = \begin{bmatrix} w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} + \begin{bmatrix} b_3 \end{bmatrix}$$

And the final activated output becomes:

$$A_2 = \begin{bmatrix} a_3 \end{bmatrix} = a(Z_2) = \begin{bmatrix} a(z_3) \end{bmatrix} = \begin{bmatrix} \frac{1}{1+e^{-z_3}} \end{bmatrix}$$

This shows a full forward pass through a 2-layer neural network. From $x \to A_1 \to A_2$, the data flows forward — layer by layer — with weights, biases, and activations applied at each stage.

**Key Insight**

Each layer is a clean block: linear operation ($Wx + b$) followed by a non-linearity ($a(z)$). Stacking these blocks is what builds a deep neural network.

## Adding More Neurons to the Second Layer

Why stop at one neuron?

We can easily add more neurons to the second layer by:

- adding more **rows** to the weight matrix $W_2$,
- and more **entries** to the bias vector $b_2$.

**Same Logic, More Neurons**

The matrix equation stays the same:

$$Z_2 = W_2 A_1 + b_2$$

Let's say we now have **three neurons** in the second layer. The output becomes:

$$A_2 = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = a(Z_2) = a\left(\begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix}\right) = \begin{bmatrix} a(z_1) \\ a(z_2) \\ a(z_3) \end{bmatrix}$$

Each $z_i$ is computed using its own weights and bias:

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \\ W_{31} & W_{32} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

**Reminder**

Each row of $W_2$ corresponds to one neuron in the layer. Each neuron gets the full input $A_1$ but transforms it differently.

## Visual Recap

Here's what we've just built — a second layer with three neurons:

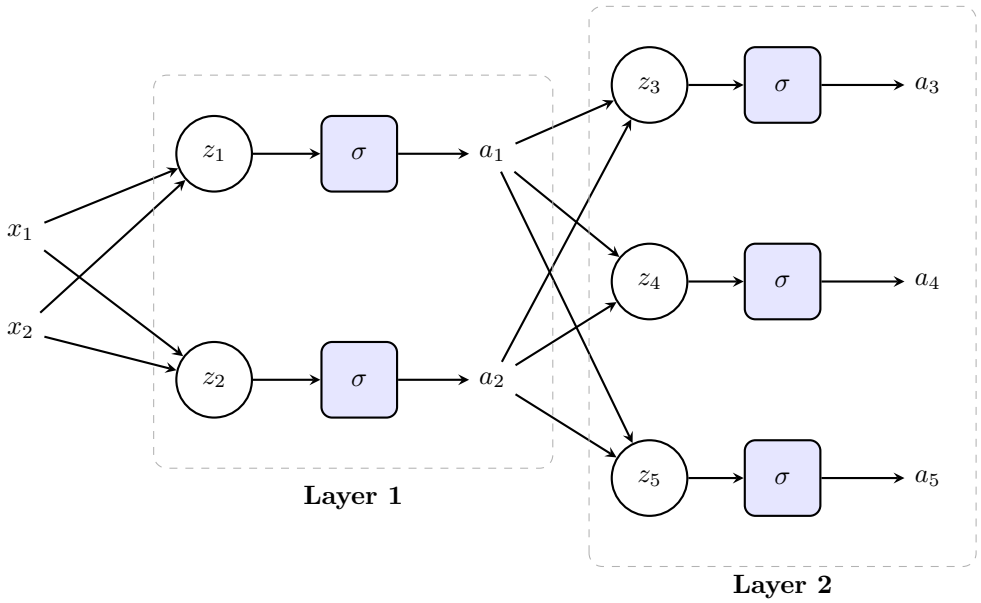



Figure 20: Two-layer neural network: 2 neurons in Layer 1, 3 neurons in Layer 2.

## Adding a Layer

**Key distinction:**

- To **add a neuron within a layer**, you simply add a new **row** to the weight matrix and a new **entry** to the bias vector.
- To **add a new layer**, you introduce an entirely new **weight matrix** and a new **bias vector**.

## Output Layer and Final Output

The **output layer** is the last layer of the neural network. It takes as input the output of the last hidden layer — and produces the final prediction.

Let's say we use the following weight matrix and bias vector:

$$W_3 = \begin{bmatrix} w_{11} & w_{12} & w_{13} \end{bmatrix} \quad \text{and} \quad b_3 = \begin{bmatrix} b_1 \end{bmatrix}$$

We compute the pre-activation of the output neuron:

$$Z_3 = \begin{bmatrix} z_1 \end{bmatrix} = W_3 A_2 + b_3 = \begin{bmatrix} w_{11} & w_{12} & w_{13} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} b_1 \end{bmatrix}$$

Then we apply the activation function (here: sigmoid) to get the final prediction:

$$A_3 = a(Z_3) = \left[\frac{1}{1+e^{-z_1}}\right]$$

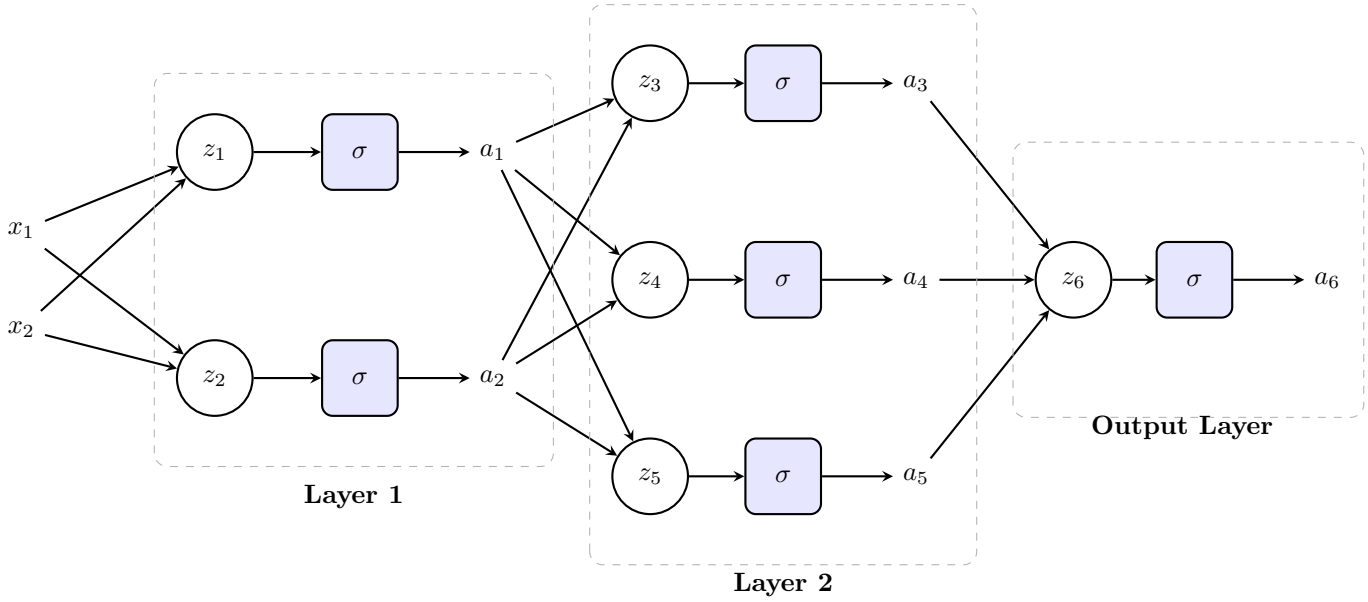



Figure 21: Complete neural network: 2 inputs, 2 hidden layers, 1 output neuron.

## Summary

- **Adding a neuron** = add a row to $W$, and an element to $b$.
- **Adding a layer** = create a new $W_i$ and $b_i$.

Each layer performs the same core sequence:

**Layer Computation — General Rule**

$$\text{Input:} \quad A_{i-1}$$
$$\text{Linear combination:} \quad Z_i = W_i A_{i-1} + b_i$$
$$\text{Activation:} \quad A_i = a(Z_i)$$

**Why matrices?** Because they allow us to compute all neurons' operations at once — fast, in parallel, and efficiently on modern hardware.

## Forward Propagation

Forward propagation is the process of passing the input through the entire network — from the input layer, through all hidden layers, to the final output.

At each step, the same logic applies:

- Apply a **linear operation** ($Z = WA + b$)
- Apply an **activation function** ($A = \sigma(Z)$)

This continues until we reach the final output:

$$A_3 = \text{network prediction}$$

## Training via Backpropagation

Once we compute the output, we want to train the network — that is, adjust the weights and biases to reduce the prediction error.

To train the network, we follow these steps:

1. Define a **loss function** (such as log-loss)
2. Compute the **gradient of the loss** with respect to each parameter
3. Update the weights and biases using **gradient descent**

From our earlier notebook (for a single neuron), we saw:

$$\frac{\partial \mathcal{L}}{\partial w_1} = -\frac{x_1}{m}\sum_i (y_i - a_i) \qquad \frac{\partial \mathcal{L}}{\partial w_2} = -\frac{x_2}{m}\sum_i (y_i - a_i) \qquad \frac{\partial \mathcal{L}}{\partial b} = -\frac{1}{m}\sum_i (y_i - a_i)$$

Remember, we derived these expressions using the chain rule.

**Now We Scale Up**

We will do the exact same thing here — but with **matrices**.

### Goal: Compute the Gradients

Let's focus on the following parameters:

- $W_1$, $b_1$ — parameters of the first layer
- $W_2$, $b_2$ — parameters of the second layer

Our goal is to understand how much each parameter influences the loss function. In other words, we want to compute:

$$\frac{\partial \mathcal{L}}{\partial W_1}, \quad \frac{\partial \mathcal{L}}{\partial b_1}, \quad \frac{\partial \mathcal{L}}{\partial W_2}, \quad \frac{\partial \mathcal{L}}{\partial b_2}$$

As before — like we did in the perceptron model — we will use the **chain rule**. This time, however, we will apply it step by step through the layers of a deep network.

**Before We Start**

**Read carefully. This is the foundation of the entire backward pass.**
You don't need to memorize equations — but you must understand the logic. It's all about how the functions are connected, and how parameters influence outputs.
If you get that, you'll understand every neural network.

## Summary of the Network Functions

**First Layer:**

- Linear transformation:
$$Z_1 = W_1 X + b_1$$
- Activation:
$$A_1 = \frac{1}{1 + e^{-Z_1}}$$

**Second Layer (Last Hidden Layer):**

- Linear transformation:
$$Z_2 = W_2 A_1 + b_2$$
*Important:* Here, the input is $A_1$, not the original input $X$.
- Activation:
$$A_2 = \frac{1}{1 + e^{-Z_2}}$$
- Loss function:
$$\mathcal{L} = -\frac{1}{m} \sum \left( y \log A_2 + (1 - y) \log(1 - A_2) \right)$$
*We compute the loss using $A_2$, the output of the last layer — not $A_1$.*

## Important

Finding the gradients means identifying how the loss $\mathcal{L}$ depends on each parameter of the network.

But as you've already seen, the **entry point** between the loss $\mathcal{L}$ and the rest of the network is the output of the last layer — namely, $A_2$.

So even if we want to compute the gradients of the parameters in the *first layer*, we must first pass through the *second layer*.

**Key Insight**

This is the core principle of backpropagation:
**We must go backward through the network — layer by layer — to compute the gradients.**

And when you think about it, it actually makes perfect sense. This is the whole point of deep learning: the model builds up its complexity layer after layer. So naturally, we need to go back down that stack to understand how each parameter affects the final outcome.

## Gradients for the Layer Just Before the Output

Let's now compute the gradients for the second layer, using the chain rule.

$$\frac{\partial \mathcal{L}}{\partial W_2} = \frac{\partial \mathcal{L}}{\partial A_2} \cdot \frac{\partial A_2}{\partial Z_2} \cdot \frac{\partial Z_2}{\partial W_2}$$

$$\frac{\partial \mathcal{L}}{\partial b_2} = \frac{\partial \mathcal{L}}{\partial A_2} \cdot \frac{\partial A_2}{\partial Z_2} \cdot \frac{\partial Z_2}{\partial b_2}$$

Let's break this down:

- $\frac{\partial \mathcal{L}}{\partial A_2}$ is the gradient of the loss with respect to the network output
- $\frac{\partial A_2}{\partial Z_2}$ is the derivative of the activation function (sigmoid)
- $\frac{\partial Z_2}{\partial W_2}$ and $\frac{\partial Z_2}{\partial b_2}$ are the local gradients of the model

Since the first two terms appear identically in both equations, we define:

$$dZ_2 = \frac{\partial \mathcal{L}}{\partial A_2} \cdot \frac{\partial A_2}{\partial Z_2}$$

Then the gradients simplify to:

$$\frac{\partial \mathcal{L}}{\partial W_2} = dZ_2 \cdot \frac{\partial Z_2}{\partial W_2}$$
$$\frac{\partial \mathcal{L}}{\partial b_2} = dZ_2 \cdot \frac{\partial Z_2}{\partial b_2}$$

## For the First Layer

Let's now focus on computing the gradients with respect to the parameters of the **first layer**.

Using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial A_1} \cdot \frac{\partial A_1}{\partial Z_1} \cdot \frac{\partial Z_1}{\partial W_1}$$

$$\frac{\partial \mathcal{L}}{\partial b_1} = \frac{\partial \mathcal{L}}{\partial A_1} \cdot \frac{\partial A_1}{\partial Z_1} \cdot \frac{\partial Z_1}{\partial b_1}$$

Where:

- $\frac{\partial \mathcal{L}}{\partial A_1}$ is the gradient of the loss with respect to the output of the first layer
- $\frac{\partial A_1}{\partial Z_1}$ is the derivative of the activation function
- $\frac{\partial Z_1}{\partial W_1}$ and $\frac{\partial Z_1}{\partial b_1}$ are the local gradients of the linear model

But — as we said before — there's a crucial observation here:

*There is no direct link between the loss $\mathcal{L}$ and the output $A_1$ of the first layer.* To compute this gradient, we must go through the second layer — just like the signal does in the forward pass.

**Pause and Think**

**How do we relate the second layer to the first?** What is the variable that connects both layers?
Take a moment and try to identify the link yourself.

The answer?
It's in the equation:

$$Z_2 = W_2 A_1 + b_2$$

This shows that $A_1$ is the input to the second layer. So to compute $\frac{\partial \mathcal{L}}{\partial A_1}$, we apply the chain rule again:

$$\frac{\partial \mathcal{L}}{\partial A_1} = \frac{\partial \mathcal{L}}{\partial A_2} \cdot \frac{\partial A_2}{\partial Z_2} \cdot \frac{\partial Z_2}{\partial A_1}$$

**What Just Happened**

We found a way to compute a gradient that had no direct path to the loss.
There is no direct dependency between $\mathcal{L}$ and $A_1$ — but by passing through $A_2$ and $Z_2$, we built the connection.
This is the heart of backpropagation: **working backward through the layers using the chain rule to connect everything.**

## Full Gradient Expression for the First Layer

Let's now write the full expression for the gradients of the first layer, step by step, with the chain rule:

$$\frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial A_2} \cdot \frac{\partial A_2}{\partial Z_2} \cdot \frac{\partial Z_2}{\partial A_1} \cdot \frac{\partial A_1}{\partial Z_1} \cdot \frac{\partial Z_1}{\partial W_1}$$

$$\frac{\partial \mathcal{L}}{\partial b_1} = \frac{\partial \mathcal{L}}{\partial A_2} \cdot \frac{\partial A_2}{\partial Z_2} \cdot \frac{\partial Z_2}{\partial A_1} \cdot \frac{\partial A_1}{\partial Z_1} \cdot \frac{\partial Z_1}{\partial b_1}$$

Let's be honest: yes, that's a long chain. But this is how deep learning works. And no, it's not "just a few lines of code." If it were simple, everyone would be doing it.

**Side Note**

Correction: everyone *is* a "data scientist" on LinkedIn. But we're here to understand what makes a **real and honest** one.

Still — we can simplify.
Just like we did earlier, we define:

$$dZ_1 = \frac{\partial \mathcal{L}}{\partial A_2} \cdot \frac{\partial A_2}{\partial Z_2} \cdot \frac{\partial Z_2}{\partial A_1} \cdot \frac{\partial A_1}{\partial Z_1}$$

But since we already defined:

$$dZ_2 = \frac{\partial \mathcal{L}}{\partial A_2} \cdot \frac{\partial A_2}{\partial Z_2}$$

We can write:

$$dZ_1 = dZ_2 \cdot \frac{\partial Z_2}{\partial A_1} \cdot \frac{\partial A_1}{\partial Z_1}$$

This gives us the final simplified expressions:

$$\frac{\partial \mathcal{L}}{\partial W_1} = dZ_1 \cdot \frac{\partial Z_1}{\partial W_1}$$
$$\frac{\partial \mathcal{L}}{\partial b_1} = dZ_1 \cdot \frac{\partial Z_1}{\partial b_1}$$

**Structure of the Backward Pass**

Each gradient is built from:

- the gradient from the next layer ($dZ_2$),
- the chain of derivatives that connects the current layer,
- and the local gradients of the linear model.

It's structured, logical, and beautifully recursive.

## Summary of the Gradient Equations

Let's recap what we want to compute during the backward pass:

$$\frac{\partial \mathcal{L}}{\partial W_2} = dZ_2 \cdot \frac{\partial Z_2}{\partial W_2}$$
$$\frac{\partial \mathcal{L}}{\partial b_2} = dZ_2 \cdot \frac{\partial Z_2}{\partial b_2}$$
$$\frac{\partial \mathcal{L}}{\partial W_1} = dZ_1 \cdot \frac{\partial Z_1}{\partial W_1}$$
$$\frac{\partial \mathcal{L}}{\partial b_1} = dZ_1 \cdot \frac{\partial Z_1}{\partial b_1}$$

To get there, we use:

$$dZ_2 = \frac{\partial \mathcal{L}}{\partial A_2} \cdot \frac{\partial A_2}{\partial Z_2}$$
$$dZ_1 = dZ_2 \cdot \frac{\partial Z_2}{\partial A_1} \cdot \frac{\partial A_1}{\partial Z_1}$$

**Strategy:**

1. Compute $dZ_2$ from the output layer.
2. Use it to compute $\frac{\partial \mathcal{L}}{\partial W_2}$ and $\frac{\partial \mathcal{L}}{\partial b_2}$.
3. Backpropagate to get $dZ_1$.
4. Use it to compute $\frac{\partial \mathcal{L}}{\partial W_1}$ and $\frac{\partial \mathcal{L}}{\partial b_1}$.

## Let's Get to the Point

In theory, things are clear. In practice, there are no shortcuts.

If we want to understand how a neural network learns, we have to compute every gradient — one by one.

**This is the work.**

And now, we're ready to do it.

## 1. Computing $dZ_2$

Let's begin with the calculation of $dZ_2$ — because this term appears in both gradient expressions for the second layer, and we also need it to backpropagate further to the first layer.

We define:

$$dZ_2 = \frac{\partial \mathcal{L}}{\partial A_2} \cdot \frac{\partial A_2}{\partial Z_2}$$

Where:

- $\frac{\partial \mathcal{L}}{\partial A_2}$ is the gradient of the loss with respect to the output of the network,

- $\frac{\partial A_2}{\partial Z_2}$ is the derivative of the sigmoid activation.

Let's compute the first term — the derivative of the loss.

$$\frac{\partial \mathcal{L}}{\partial A_2} =?$$

**Wait... you don't remember this?**

Come on — it's the same expression we derived earlier in the previous notebook using the log-loss function. Fine. I'll recap it but after that, you're on your own. I'm not your mom. For goodness' sake.

**Loss function (log-loss):**

$$\mathcal{L} = -\frac{1}{m}\sum_i \left(y_i \log A_2 + (1-y_i)\log(1-A_2)\right)$$

We compute its derivative with respect to $A_2$:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial A_2} &= -\frac{1}{m}\sum_i \left(y_i \cdot \frac{1}{A_2} - (1-y_i)\cdot \frac{1}{1-A_2}\right) \\
&= -\frac{1}{m}\sum_i \left(\frac{y_i}{A_2} - \frac{1-y_i}{1-A_2}\right) \\
&= -\frac{1}{m}\sum_i \left(\frac{y_i(1-A_2) - (1-y_i)A_2}{A_2(1-A_2)}\right) \\
&= -\frac{1}{m}\sum_i \left(\frac{y_i - A_2}{A_2(1-A_2)}\right)
\end{aligned}$$

So yes — this is the final simplified form:

$$\boxed{\frac{\partial \mathcal{L}}{\partial A_2} = -\frac{1}{m}\sum_i \left(\frac{y_i - A_2}{A_2(1-A_2)}\right)}$$

We now compute the second term in:

$$dZ_2 = \frac{\partial \mathcal{L}}{\partial A_2}\cdot\frac{\partial A_2}{\partial Z_2}$$

Namely:

$$\frac{\partial A_2}{\partial Z_2} = A_2\cdot(1-A_2)$$

*No way... you forgot this one too?*

All right — I said I wouldn't hold your hand anymore, but let's be honest — this notebook is meant to be your guide.

So let's recap.

The activation function introduces non-linearity into the network, allowing it to model complex relationships between inputs and outputs.

The sigmoid activation is defined by:

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

We now compute its derivative:

$$\begin{aligned}
\frac{d}{dz}\left(\frac{1}{1+e^{-z}}\right) &= \frac{-1}{(1+e^{-z})^2} \cdot \frac{d}{dz}(1+e^{-z}) \\
&= \frac{-1}{(1+e^{-z})^2} \cdot (-e^{-z}) \\
&= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{1}{1+e^{-z}} \cdot \left(\frac{e^{-z}}{1+e^{-z}}\right) \\
&= a(z) \cdot (1 - a(z))
\end{aligned}$$

So for any neuron output $z$, the derivative becomes:

$$\boxed{\frac{da(z)}{dz} = a(z)(1 - a(z))}$$

Hence, for the entire vector $A_2$, we apply this element wise:

$$\frac{\partial A_2}{\partial Z_2} = \begin{bmatrix} a(z_1)(1-a(z_1)) \\ a(z_2)(1-a(z_2)) \\ a(z_3)(1-a(z_3)) \end{bmatrix} \quad \text{or simply:} \quad A_2 \cdot (1 - A_2)$$

**Final Expression for $dZ_2$**

We now write the full expression for the error signal at the output layer:

$$\begin{aligned}
dZ_2 &= \frac{\partial \mathcal{L}}{\partial A_2} \cdot \frac{\partial A_2}{\partial Z_2} \\
&= -\frac{1}{m} \sum_i \left(\frac{y_i - A_2}{A_2(1 - A_2)}\right) \cdot A_2 \cdot (1 - A_2) \\
&= -\frac{1}{m} \sum_i (y - A_2) \\
&= A_2 - y
\end{aligned}$$

**Why is $\frac{1}{m}\sum_i (A_2 - y) = A_2 - y$ legitimate here?**

Because we are, at this stage, working with a single training example. In that context, summing over $i$ yields only one term — so the average is the term itself. No division by $m$ is needed.

*However:* the averaging over multiple samples is not forgotten. **It will be accounted for during parameter updates.**

**Result — Output Layer Error**

$$\boxed{dZ_2 = A_2 - y}$$

This expression — the difference between prediction and truth — is the entry point of the backward pass.

## 2. $\frac{\partial \mathcal{L}}{\partial W_2}$ and $\frac{\partial \mathcal{L}}{\partial b_2}$

Once $dZ_2$ is computed, we can express the gradients of the loss with respect to the parameters of the second layer:

$$\boxed{\frac{\partial \mathcal{L}}{\partial W_2} = dZ_2 \cdot \frac{\partial Z_2}{\partial W_2}} \quad \text{and} \quad \boxed{\frac{\partial \mathcal{L}}{\partial b_2} = dZ_2 \cdot \frac{\partial Z_2}{\partial b_2}}$$

> I want you to look at the different equations we highlighted before to understand the relation between the different variables and find the wanted gradients.

If you look at the equations, you will see that we can find these gradients using the second layer model:

$$Z_2 = W_2 A_1 + b_2$$

We will now compute each of these terms explicitly.

$$\frac{\partial Z_2}{\partial W_2} = A_1$$

where $A_1$ is the output of the previous layer.

Similarly, we write the expression for the bias term:

$$\frac{\partial Z_2}{\partial b_2} = 1$$

This corresponds to the derivative of the affine shift with respect to its own scalar.

Substituting into the chain rule:

$$\frac{\partial \mathcal{L}}{\partial W_2} = dZ_2 \cdot A_1$$
$$\frac{\partial \mathcal{L}}{\partial b_2} = dZ_2 \cdot 1$$

Where:

- $dZ_2 = A_2 - y$ is the gradient from the output layer,
- $A_1$ is the activation of the previous layer.

Now we recover the factor $\frac{1}{m}$ that we omitted earlier:

- The summation is implicitly handled by the matrix product $dZ_2 \cdot A_1^T$, since we are using vectorized notation.
- For the bias — which is scalar or vector-valued per neuron — we sum the components of $dZ_2$ explicitly.

**Thus, the final expressions are:**

$$\boxed{\frac{\partial \mathcal{L}}{\partial W_2} = \frac{1}{m} (A_2 - y) \cdot A_1^\top}$$

$$\boxed{\frac{\partial \mathcal{L}}{\partial b_2} = \frac{1}{m} \sum_m (A_2 - y)}$$

## 3. Computing $dZ_1$

We now compute $dZ_1$, which is required to obtain the gradients of the first layer:

$$\frac{\partial \mathcal{L}}{\partial W_1} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial b_1}$$

The expression for $dZ_1$ comes from the chain rule applied backward:

$$dZ_1 = dZ_2 \cdot \frac{\partial Z_2}{\partial A_1} \cdot \frac{\partial A_1}{\partial Z_1}$$

Where:

- $dZ_2$ is the gradient of the loss with respect to the pre-activation output $Z_2$,
- $\frac{\partial Z_2}{\partial A_1} = W_2$ — from the layer equation $Z_2 = W_2 A_1 + b_2$,
- $\frac{\partial A_1}{\partial Z_1} = A_1 \cdot (1 - A_1)$ — the derivative of the sigmoid activation.

So we write:

$$\boxed{dZ_1 = (dZ_2 \cdot W_2) \circ A_1 \cdot (1 - A_1)}$$

where $\circ$ denotes the element-wise (Hadamard) product.

**Interpretation**

We are propagating the gradient from layer 2 through the weights $W_2$, modulating it by the sensitivity of the activations $A_1$ — just as we did in the output layer.

## 4. $\frac{\partial \mathcal{L}}{\partial W_1}$ and $\frac{\partial \mathcal{L}}{\partial b_1}$

Now with $dZ_1$, we compute:

$$\frac{\partial \mathcal{L}}{\partial W_1} = dZ_1 \cdot \frac{\partial Z_1}{\partial W_1}$$
$$\frac{\partial \mathcal{L}}{\partial b_1} = dZ_1 \cdot \frac{\partial Z_1}{\partial b_1}$$

From the model:

$$Z_1 = W_1 X + b_1$$

we derive:

$$\frac{\partial Z_1}{\partial W_1} = X \quad \text{and} \quad \frac{\partial Z_1}{\partial b_1} = 1$$

Where:

- $X$ is the input data matrix,
- 1 represents a constant vector aligned with the bias shape.

We can now express the raw gradients:

$$\frac{\partial \mathcal{L}}{\partial W_1} = dZ_1 \cdot X$$
$$\frac{\partial \mathcal{L}}{\partial b_1} = dZ_1 \cdot 1$$

with $dZ_1$ being the gradient of the loss with respect to the pre-activation values $Z_1$.

**As before**, we must include the normalization factor $\frac{1}{m}$, and sum over samples for the bias term.

**Final vectorized expressions:**

$$\boxed{\begin{aligned} \frac{\partial \mathcal{L}}{\partial W_1} &= \frac{1}{m} dZ_1 \cdot X^\top \\ \frac{\partial \mathcal{L}}{\partial b_1} &= \frac{1}{m} \sum_m dZ_1 \end{aligned}}$$

Where:

$$dZ_1 = (dZ_2 \cdot W_2) \circ A_1 \cdot (1 - A_1)$$

is the gradient of the loss function with respect to the output of the first layer.

**Key Insight**

All gradients are now expressed as matrix products or elementwise operations — enabling efficient and scalable backpropagation.

## Summary of the Gradients

Here it is.

No more equations, no more calculus — the math is done. We now have everything we need to update the parameters of our neural network.

**Final gradient expressions:**

$$\boxed{\begin{aligned}\frac{\partial \mathcal{L}}{\partial W_2} &= \frac{1}{m}(A_2 - y) \cdot A_1^\top \\ \frac{\partial \mathcal{L}}{\partial b_2} &= \frac{1}{m}\sum_m (A_2 - y) \\ \frac{\partial \mathcal{L}}{\partial W_1} &= \frac{1}{m} dZ_1 \cdot X^\top \\ \frac{\partial \mathcal{L}}{\partial b_1} &= \frac{1}{m}\sum_m dZ_1\end{aligned}}$$

**Intermediate expressions:**

$$\boxed{\begin{aligned}dZ_2 &= A_2 - y \\ dZ_1 &= (dZ_2 \cdot W_2) \circ A_1 \cdot (1 - A_1)\end{aligned}}$$

*I know what you're thinking:*

"Those equations are ugly as hell."

But here's the truth:

1. I don't care — this is the real stuff. We're not at a fashion show.
2. You only need to do this **once**.

A data scientist isn't a mathematician. They don't compute gradients by hand. They build systems, they automate, they abstract.

**But** — they need to understand what happens under the hood. That's why you've done the work.

And now, you're ready.

You've gone through every step. No shortcuts, no magic. Just honest computation.

**Next Stop — Practice**

Let's code this network from scratch. It'll be shockingly easy — because the math is already done.

Meet me in:

`practice_02.ipynb`

practice_02.ipynb

**Launch `practice_02.ipynb` on Google Colab**

*Follow this part on Google Colab here.*
*Train, visualize decision boundaries, understand non-linearity.*
*Change the parameters and see the consequences by yourself on this pre-coded playground.*

# Practice: Building a Two-Layer Neural Network

## A Neural Network Is Just a Structured Neuron

We construct a neural network just like we did with a single neuron.
And that's the most powerful insight you can carry forward:

> *If you understand the structure of a single neuron, you can understand the structure of a full network.*

A neuron needs:

- A model to compute outputs,
- A cost function to measure performance,
- An optimization rule to improve it,
- And a method to update parameters.

So does a neural network.

## What Remains the Same?

Surprisingly: **almost everything.**

- Same architectural flow: `model → cost → optimization → update`
- Same activation function
- Same cost function
- Same rule for updating parameters

Only one thing changes — and it changes everything.

### What Changes: Dimensionality

When you scale up to multiple layers, a new concept enters the stage:

## Dimensionality

It's the foundation of neural network design.

With just one neuron, dimensions are trivial. But when you have several layers and many neurons per layer, you need to define:

- The number of layers
- The number of neurons per layer

And that's not just for looks.

### Why Dimensionality Matters

Because neural networks are built with **matrices**.

And with matrices, everything depends on compatibility of dimensions:

1. **Correct dimensions = valid matrix calculations.** No match, no forward pass.
2. **Matrix operations are the core of computation.** They power your model.
3. **Architecture must match the problem.** Structure is what transforms input into insight.

### How We Use It

You define the architecture of a neural network at the moment you **initialize the parameters** — that is, the weights and biases.

**Core Principle**

Creating a new layer? → Define a new weight matrix and a bias vector.
Adding more neurons? → Increase the number of rows in that matrix.

From here, every layer is just a new transformation — and each transformation is built by the dimensions you choose.

## Step-by-Step: Building the Network Layer by Layer

**For each layer**, you must:

- Define a weight matrix and a bias vector,
- Choose the number of neurons in the layer,
- Ensure that the matrix dimensions match the input/output of the surrounding layers.

But there's a catch:

> Your network must **match the input dimension of your dataset**.

### Example: Toxic vs. Non-Toxic Plants

In the first notebook example, each plant is described by two features:

- Leaf length
- Leaf width

So your input data has **2 dimensions**. Each plant is a point in a 2D space — one axis for length, one for width.

**Implication:** Your first layer's weight matrix must reflect these 2 input features.

> Don't get stuck on "rows vs. columns." What matters is **compatibility** between your input data and the weight matrix.

If your weight matrix has:

- 2 rows (for the two features),
- and as many columns as neurons in the layer,

then great — it works.

If you accidentally flip that — no panic.

> Just transpose the matrix with `.T`. It's mathematically equivalent. If you're unfamiliar with matrix transposition, it's a quick Google away.

### What Matters Is Shape Compatibility

The shape of each weight matrix must match:

- The number of **input features** it receives,
- The number of **neurons** it sends outputs to.

Once you choose a convention (e.g., rows = features, columns = neurons), **stick to it across the whole network.**

> *Consistency in dimensions = seamless matrix calculations = a working neural network.*

The bias vector for each layer simply matches the number of neurons in that layer — i.e., the number of columns in the weight matrix.

### And Then What?

For each subsequent layer:

- The number of **rows** in the new weight matrix = number of neurons in the previous layer,
- The number of **columns** = number of neurons in the current layer.

You repeat this process until the final layer.

In the output layer, the number of columns equals the number of outputs you want:

- **Binary classification** → 1 output neuron
- **Multi-class classification** → one neuron per class
- **Clustering** → one neuron per cluster

*(Yeah, I told you you'd be a data scientist by the end of this journey.)*

**Key Takeaway**

Proper parameter dimensions allow layers to connect and communicate via matrix operations. That's the core purpose of initialization: **not to fill random values**, but to define a functional, adaptable architecture aligned with your problem.

## If You Got This Far

Let me be honest for a second.

**Congratulations** — you now understand something many people spend weeks struggling with.
In my engineering school (I'm in M2, Data Science), half of my classmates still don't fully grasp this.
*Translation:* 50% of my cohort doesn't really know what a neural network is.

You, on the other hand, do.
You're already ahead of the game. You're entering an exclusive circle of people who genuinely understand how deep learning works — not by imitation, but by structure.

**Ready for the Next Step**

Let's now move from understanding to implementation.
It's time to code all of this.

## Code

## DataSet

You are a data scientist. But more than that — you are an explorer.

One day, you discover a mysterious land on planet Earth. There's only one thing to eat: plants. Some of your fellow explorers have already gotten sick. The culprit? They ate the wrong plants. You decide to take action. Your mission: **predict whether a plant is toxic or not**.
You've compiled a list of plants and recorded two simple features:

- Leaf length
- Leaf width

Then, like any good scientist, you plot the data. Here's what you see:

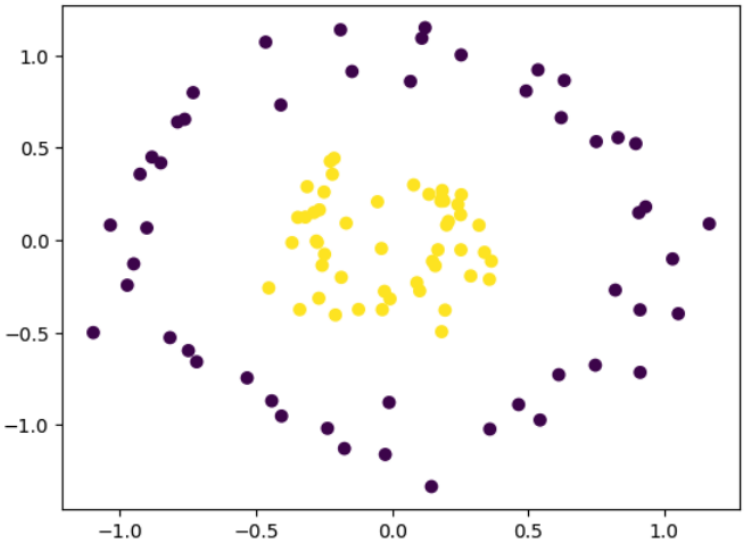



Figure 22: Toxic vs. Non-Toxic Plants

From the graph, one thing becomes crystal clear:

**A linear model can't do the job.**

The boundary between toxic and non-toxic plants is clearly nonlinear. So what can we do?

## Machine Learning

In classical machine learning, one popular strategy is **feature engineering**. That means creating new features based on the existing ones — for example:

- The product $x_1 \cdot x_2$
- The square $x_1^2$ or $x_2^2$

By doing this, we might capture some hidden structure in the data and help a linear model make sense of it.
But here's the catch:

> *Feature engineering is an art. It's time-consuming, error-prone, and often domain-specific.*

### Deep Learning

That's exactly where deep learning shines.

Rather than crafting features by hand, we let the network **learn its own internal features**. Layer by layer, it transforms the input data into increasingly meaningful representations.

Even more amazing?

**Each neuron still performs a linear operation.**

What creates the magic is the **composition of many layers**, each followed by a **nonlinear activation**. That's how the model bends space, finds patterns, and builds its own intelligence.

So yes — what you're seeing is more than a model.

It's the beginning of a machine that learns to understand the world.

## Initialization

Before our network can learn, we must give it structure. That means initializing its parameters — the weights and biases that govern how signals move through the layers.

To do this, we need just two ingredients:

1. The **input dimension** $n_0$ — here, that's 2 (leaf length and width)
2. The **number of neurons per layer** — for example, $n_1$ for the first (hidden) layer, and $n_2$ for the output

With these, we define our parameters:

- $\boldsymbol{W^{[1]}}$ — the weight matrix from input to hidden layer
- $\boldsymbol{b^{[1]}}$ — the bias vector of the hidden layer
- $\boldsymbol{W^{[2]}}$ — the weight matrix from hidden to output
- $\boldsymbol{b^{[2]}}$ — the bias of the output

All these parameters will be stored in a single dictionary, so we can pass them around and update them during training.

Let's see how we set this up in code.

```python
def initialisation(n0, n1, n2):
    # n0 is the input dimension     number of features going into the network
    # n1 is the number of neurons in the first (hidden) layer
    # n2 is the output dimension     number of neurons in the output layer

    # First layer: connects input features to the hidden layer
    W1 = np.random.randn(n1, n0)  # weights matrix (n1 neurons    n0 inputs)
    b1 = np.random.randn(n1, 1)   # bias vector for hidden layer (n1    1)

    # Second layer: connects hidden layer to output
    W2 = np.random.randn(n2, n1)  # weights matrix (n2 neurons    n1 inputs)
    b2 = np.random.randn(n2, 1)   # bias vector for output layer (n2    1)

    # Bundle everything in a dictionary for easy access
    parameters = {
        'W1': W1,
```

```
        'b1': b1,
        'W2': W2,
        'b2': b2
    }

    return parameters
```

Let's take a closer look.
We want to classify each plant as either **toxic** or **non-toxic**. That makes it a:

**Binary classification problem.**

So what does that imply?

> We only need **one output neuron**.
> So we set: $\boldsymbol{n_2 = 1}$

Now let's look at our input data.
Each plant is described by two features:

- Leaf length
- Leaf width

So:

> The input dimension is $\boldsymbol{n_0 = 2}$

That leaves us with one open question: How many neurons should we use in the hidden layer?

> This is a design choice — a hyperparameter.
> For now, let's try with: $\boldsymbol{n_1 = 4}$.

This gives the network enough capacity to learn non-linear patterns, while staying lightweight enough to understand every step of the process.
We are now ready to move forward — and build the forward pass of our neural network.

## Forward Propagation

You've just mastered the most important shift from a single neuron to a full neural network: **initialization**.
That's where everything changes — we now have multiple layers, each with its own weights and biases. But here's the good news:

> The rest of the functions stay conceptually the same. Only one thing changes: **we go from scalars to matrices**.

That's right — matrix computation is now the **backbone** of everything we do.

Now that the network is initialized, it's time to **send the input data through the layers**. This process is called:

# Forward Propagation

Its goal is to compute the **activations** — the outputs — of every neuron in every layer. Just like before, each neuron will apply:

- A **linear transformation**: $Z = W \cdot X + b$
- Followed by a **nonlinear activation**: $A = \sigma(Z)$

So for our two-layer network, the full forward propagation is:

$$
\begin{aligned}
Z^{[1]} &= W^{[1]} X + b^{[1]} \\
A^{[1]} &= \sigma(Z^{[1]}) \\
Z^{[2]} &= W^{[2]} A^{[1]} + b^{[2]} \\
A^{[2]} &= \sigma(Z^{[2]}) = \hat{y}
\end{aligned}
$$

Where:

- $X$ is the input data (shape: $n_0 \times m$)
- $Z^{[1]}, Z^{[2]}$ are the linear outputs (pre-activations)
- $A^{[1]}, A^{[2]}$ are the activations
- $\sigma(\cdot)$ is the sigmoid activation function
- $\hat{y}$ is the model's prediction for each example

And remember: all of this is vectorized. Which means it's not just fast — it's elegant.

We're no longer thinking neuron-by-neuron. We're thinking in **layers**, in **patterns**, in **abstractions**.

That's what makes deep learning so powerful.

Are you ready to implement this in code?

```python
def forward_propagation(X, parameters):
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']

    # First hidden layer
    Z1 = W1.dot(X) + b1             # Linear combination: Z1 = W1 X + b1
    A1 = 1 / (1 + np.exp(-Z1))      # Activation using sigmoid

    # Output layer
    Z2 = W2.dot(A1) + b2            # Linear combination: Z2 = W2 A1 + b2
    A2 = 1 / (1 + np.exp(-Z2))      # Final sigmoid activation

    activations = {
        'A1': A1,
        'A2': A2
    }

    return activations
```

After running this function, we obtain the activations of all the neurons in our network, neatly stored in the `activations` dictionary.

Let's unpack what each variable means:

- $\boldsymbol{A^{[1]}}$ is the activation of the **first hidden layer**. Since this layer has $n_1$ neurons, $A^{[1]}$ has a shape of $(n_1, m)$ — one activation per neuron, for each of the $m$ training examples.
- $\boldsymbol{A^{[2]}}$ is the **final output** of the network — our prediction $\hat{y}$. As we're doing binary classification, this is a single probability per example. Shape: $(1, m)$

In summary:

> This function transforms the raw input data $X$ into high-level predictions $\hat{y}$, layer by layer. All by stacking linear steps and nonlinear activations.

And it's all fully vectorized — making it blazing fast, even for large datasets.

## Loss Function

Now that we have the model's predictions $\hat{y}$ from forward propagation, we need a way to evaluate how good these predictions are.

This is the role of the **loss function**.

In binary classification problems, the most common choice is the **log loss** (also called binary cross-entropy):

$$\mathcal{L} = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(\hat{y}^{(i)}) + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right]$$

But instead of implementing this manually, we'll use a built-in function from `scikit-learn`, which handles everything — including numerical stability.

```
from sklearn.metrics import log_loss

# Compute the loss
loss = log_loss(y_true=y.flatten(), y_pred=activations['A2'].flatten())

print("Log Loss:", loss)
```

`log_loss` automatically compares:

- the `true labels` $y$ (which must be 1D: that's why we use `.flatten()`)
- and the `predicted probabilities` $\hat{y} = A^{[2]}$

This gives us a scalar: the average penalty the model pays for its incorrect predictions.

> The lower the log loss, the better the predictions.

We now have a full forward pass: from input to prediction to loss.

Next step? We need to teach the model how to get better — by minimizing this loss.

Let's move to the **backward propagation**.

## Backward Propagation

Now that we know how to compute the loss — the model's error — the next question is:

> *How do we update the parameters to make the model better?*

That's where the magic happens: We go backward through the network and compute the gradients of the loss with respect to each parameter.

This is called:

# Backward Propagation

Here's the logic:

- We compute the gradient of the loss with respect to the output $\hat{y}$ (from the output layer)
- Then we propagate this error backward through the network using the chain rule
- At each layer, we compute the gradients with respect to:
  - The weights $W$
  - The biases $b$
  - And the activations from the previous layer

Remember that the gradients we found using the **chain rule** :

$$\begin{cases} \dfrac{\partial \mathcal{L}}{\partial \mathbf{W_2}} = \dfrac{\mathbf{1}}{\mathbf{m}} (\mathbf{A_2} - \mathbf{y}) \cdot \mathbf{A_1^T} \\ \dfrac{\partial \mathcal{L}}{\partial \mathbf{b_2}} = \dfrac{\mathbf{1}}{\mathbf{m}} \sum_{\mathbf{m}} (\mathbf{A_2} - \mathbf{y}) \\ \dfrac{\partial \mathcal{L}}{\partial \mathbf{W_1}} = \dfrac{\mathbf{1}}{\mathbf{m}} \mathbf{dZ_1} \cdot \mathbf{X^T} \\ \dfrac{\partial \mathcal{L}}{\partial \mathbf{b_1}} = \dfrac{\mathbf{1}}{\mathbf{m}} \sum_{\mathbf{m}} \mathbf{dZ_1} \end{cases}$$

With the terms:

$$\begin{cases} \mathbf{dZ_2} = \mathbf{A_2} - \mathbf{y} \\ \mathbf{dZ_1} = \mathbf{dZ_2} \cdot \mathbf{W_2} \cdot \mathbf{A_1} \cdot (\mathbf{1} - \mathbf{A_1}) \end{cases}$$

Let's write this step-by-step for our 2-layer network, using vectorized operations.

```python
def backward_propagation(X, y, parameters, activations):
    m = X.shape[1]  # number of examples

    # Retrieve parameters and activations
    W2 = parameters['W2']
    A1 = activations['A1']
    A2 = activations['A2']

    # Compute gradients of the loss w.r.t. output (A2)
    dZ2 = A2 - y
```

```
    dW2 = (1 / m) * dZ2.dot(A1.T)
    db2 = (1 / m) * np.sum(dZ2, axis=1, keepdims=True)

    # Propagate back to the hidden layer
    dZ1 = W2.T.dot(dZ2) * A1 * (1 - A1)
    dW1 = (1 / m) * dZ1.dot(X.T)
    db1 = (1 / m) * np.sum(dZ1, axis=1, keepdims=True)

    gradients = {
        'dW1': dW1,
        'db1': db1,
        'dW2': dW2,
        'db2': db2
    }

    return gradients
```

What just happened?

- $\boldsymbol{dZ^{[2]} = A^{[2]} - y}$ is the error at the output
- $\boldsymbol{dW^{[2]}}$ and $\boldsymbol{db^{[2]}}$ are the gradients for the second layer
- We then propagate the error to the hidden layer using the derivative of the sigmoid:

$$\sigma'(Z) = \sigma(Z) \cdot (1 - \sigma(Z)) = A \cdot (1 - A)$$

- Finally, we compute the gradients $\boldsymbol{dW^{[1]}}$ and $\boldsymbol{db^{[1]}}$

This backward pass gives us the full gradient information we need to improve the model.
All that remains is to use these gradients to update the weights.
Let's move to the final step: **gradient descent**.

## Gradient Descent: Updating the Parameters

We've come a long way.
At this point, we have:

- Initialized the network
- Performed forward propagation to make predictions
- Computed the loss to evaluate performance
- Performed backward propagation to compute gradients

There's only one thing left to do:

**Update the parameters to improve the model.**

This is done using the most classic algorithm in machine learning: **Gradient Descent**.

At each step, we move the weights and biases slightly in the direction that reduces the loss. We do this by subtracting the gradient, scaled by a factor called the **learning rate** $\alpha$.

**Mathematically:**

$$W^{[1]} \leftarrow W^{[1]} - \alpha \cdot dW^{[1]}$$
$$b^{[1]} \leftarrow b^{[1]} - \alpha \cdot db^{[1]}$$
$$W^{[2]} \leftarrow W^{[2]} - \alpha \cdot dW^{[2]}$$
$$b^{[2]} \leftarrow b^{[2]} - \alpha \cdot db^{[2]}$$

Let's implement it in code.

```python
def update_parameters(parameters, gradients, learning_rate):
    parameters['W1'] -= learning_rate * gradients['dW1']
    parameters['b1'] -= learning_rate * gradients['db1']
    parameters['W2'] -= learning_rate * gradients['dW2']
    parameters['b2'] -= learning_rate * gradients['db2']

    return parameters
```

That's it. This function takes the current parameters, the gradients from backpropagation, and the learning rate — and returns updated weights and biases.

This is the heartbeat of learning:
**Forward → Loss → Backward → Update**.
Repeat this loop, and the model gets better.

Now, we're ready to train the full model.
Let's bring everything together in a training loop.

## Training the Neural Network

We now have all the building blocks to train our model.

Let's summarize the full training process:

1. Initialize the parameters
2. Repeat for a number of iterations:
    - Forward propagation to get predictions
    - Compute the loss
    - Backward propagation to get gradients
    - Update parameters using gradient descent

Let's implement this in a function. Before diving into the code, let's clarify a key idea:

Our model outputs a probability $\hat{y}$ between 0 and 1, thanks to the sigmoid activation.

But we ultimately want a binary prediction: **toxic** or **non-toxic** — 1 or 0.
That's why we apply a threshold:

```
predictions = (A2 > 0.5)
```

In other words:

- If $\hat{y} > 0.5 \rightarrow$ predict class 1 (toxic)
- If $\hat{y} \leq 0.5 \rightarrow$ predict class 0 (non-toxic)

This gives us actual class predictions, which we can compare to the ground truth to compute accuracy.

Here's the full training function — now named `neural_network`:

```python
def neural_network(X, y, n1=4, iterations=1000, learning_rate=0.1, verbose=True):
    from sklearn.metrics import log_loss, accuracy_score

    n0 = X.shape[0]   # Input dimension
    n2 = 1            # Binary output

    parameters = initialisation(n0, n1, n2)
    losses = []
    accuracies = []

    for i in range(iterations):
        # Forward pass
        activations = forward_propagation(X, parameters)
        A2 = activations['A2']

        # Loss
        loss = log_loss(y.flatten(), A2.flatten())
        losses.append(loss)

        # Binary predictions based on 0.5 threshold
        predictions = (A2 > 0.5)

        # Accuracy
        acc = accuracy_score(y.flatten(), predictions.flatten())
        accuracies.append(acc)

        # Backward pass
        gradients = backward_propagation(X, y, parameters, activations)

        # Update parameters
        parameters = update_parameters(parameters, gradients, learning_rate)

        # Print progress
        if verbose and i % 100 == 0:
            print(f"Iteration {i}     Loss: {loss:.4f}     Accuracy: {acc:.2%}")

    return parameters, losses, accuracies
```

This function returns everything you need to evaluate and visualize your model's learning journey.

By turning probabilities into decisions, we turn predictions into action.

This function takes your input data $X$, the labels $y$, and some training hyperparameters:

- Number of neurons in the hidden layer: `n1`

- Number of iterations
- Learning rate

It returns:

- The final parameters (trained weights and biases)
- The list of loss values over time — so you can monitor training progress

Now, let's run this function with 32 neuron in the hidden layer and visualize the result:

```
parameters, losses = neural_network(X, y, n1=32, iterations=1000, learning_rate
    =0.1)
```

This gives us a clear view of how the model improves over time.

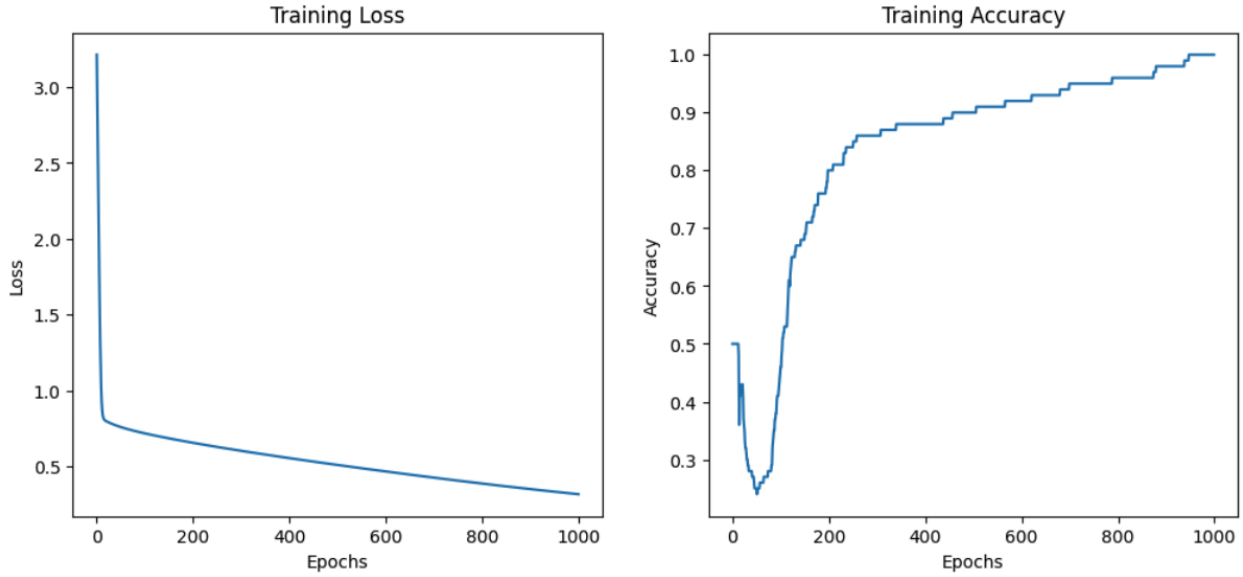



Figure 23: Training Loss and Accuracy

## Interpreting the Training Curves

- **Left: Log Loss**
  The curve drops sharply in the early iterations — a sign that the model is learning quickly.
  Then it decreases more slowly but steadily, as the model refines its predictions.
  Once the loss goes below 0.5, we know it has captured the underlying structure of the data.

- **Right: Accuracy**
  We observe a small dip at the beginning — this is expected as the model starts with random weights.
  Then accuracy rises progressively, eventually approaching 100% — an excellent result.
  The sharp climb around iteration 100 marks the moment the network begins to truly "understand".

Together, these curves confirm that the network is effectively learning a non-linear decision boundary from the data.

We've now built a complete neural network from scratch — and trained it to make intelligent predictions on nonlinear data.

Next challenge? Let's visualize the decision boundary and see how well the model separates toxic from non-toxic plants.

## Remember the Decision Boundary?

The decision boundary is the set of input points where the model is perfectly uncertain — it doesn't know whether to classify the input as class 0 or class 1.

In binary classification, this happens when:

$$y_{\text{pred}} = 0.5 \quad \Leftrightarrow \quad A^{[2]} = 0.5 \quad \Leftrightarrow \quad Z^{[2]} = 0 \quad \Leftrightarrow \quad W^{[2]}A^{[1]} + b^{[2]} = 0$$

In other words:

**The decision boundary is where the neural network hesitates.**

In our model, forward propagation follows these steps:

- $Z^{[1]} = W^{[1]}X + b^{[1]}$
- $A^{[1]} = \sigma(Z^{[1]})$
- $Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$
- $A^{[2]} = \sigma(Z^{[2]}) = \hat{y}$

Where $X = (x_1, x_2)$ is the input data.

## How Do We Compute the Decision Boundary?

We want to find the set of input points $(x_1, x_2)$ such that $\hat{y} = 0.5$.

To visualize this boundary:

1. Create a grid of input points $(x_1, x_2)$ over the input space
2. Use `forward_propagation` to compute $\hat{y}$ on each point
3. Plot the region where $\hat{y} = 0.5$

Here's the intuition:

- $\hat{y} < 0.5 \rightarrow$ model predicts class 0
- $\hat{y} > 0.5 \rightarrow$ model predicts class 1
- $\hat{y} = 0.5 \rightarrow$ the decision boundary

## Why is the Decision Boundary Non-Linear?

Even though each layer in a neural network performs a linear operation followed by a sigmoid, the stacking of layers introduces non-linearity:

- The hidden layer transforms the input space non-linearly
- The output layer applies a new transformation on that space

The composition of these functions allows the network to **bend and twist** the input space, producing highly complex decision regions.

> That's the core of deep learning: **learning internal representations without manual feature engineering.**

## Summary

- The decision boundary corresponds to points where $\hat{y} = 0.5$
- We find it by evaluating the model on a grid of inputs
- Neural networks learn complex, nonlinear boundaries through stacked layers
- This removes the need for manual feature engineering

This is the power of deep learning: **learning boundaries directly from data, no feature engineering required.**

```python
def plot_decision_boundary(X, y, parameters):
    # Create a grid of points
    x_min, x_max = X[0, :].min() - 0.5, X[0, :].max() + 0.5
    y_min, y_max = X[1, :].min() - 0.5, X[1, :].max() + 0.5
    h = 0.01  # Step size for the mesh

    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    # Flatten and stack to get shape (2, number of points)
    grid = np.c_[xx.ravel(), yy.ravel()].T

    # Predict on the grid
    activations = forward_propagation(grid, parameters)
    Z = activations['A2']
    Z = Z.reshape(xx.shape)

    # Plot the contour and training examples
    plt.figure(figsize=(7, 6))
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral, alpha=0.7)
    plt.scatter(X[0, :], X[1, :], c=y.flatten(), cmap=plt.cm.Spectral, edgecolors='k')
    plt.title("Decision Boundary of the Trained Neural Network")
    plt.xlabel("Leaf Length")
    plt.ylabel("Leaf Width")
    plt.grid(True)
    plt.show()
```

Let's plot it:

```
plot_decision_boundary(X, y, parameters)
```

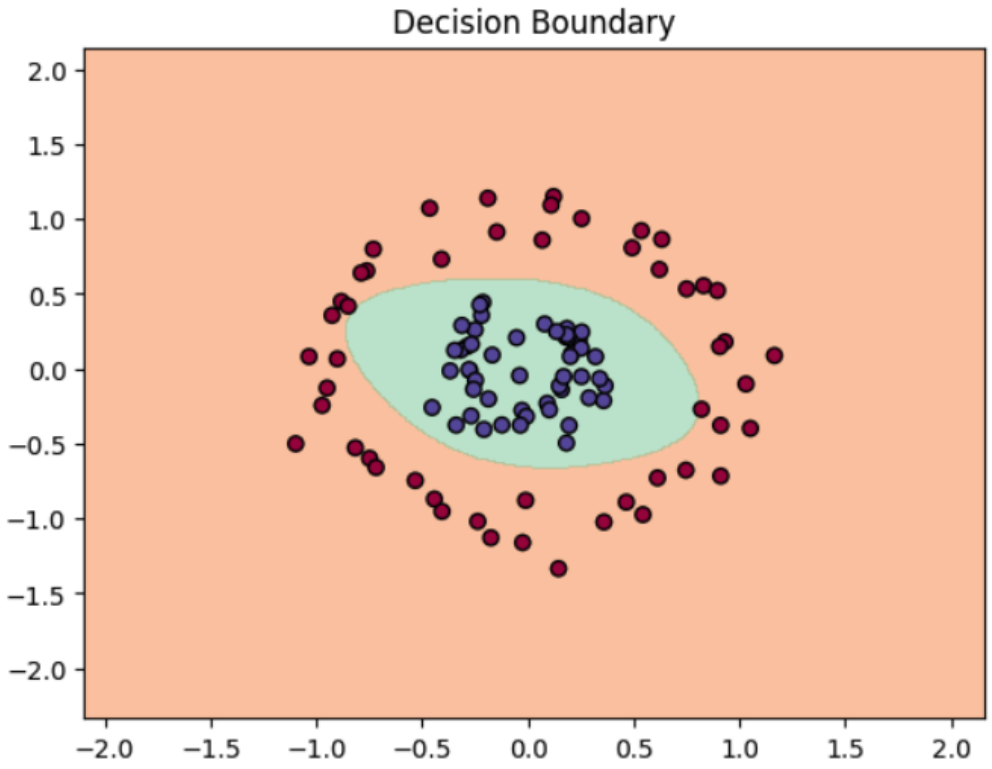



Figure 24: Decision Boundary of the Neural Network

## Decision Boundary: What Happens When We Change the Training Setup?

The figure below shows the decision boundary produced by our neural network with:

- **32 neurons** in the hidden layer
- A **learning rate** of 0.1
- Trained for **1000 epochs**

We clearly observe a **non-linear decision boundary** that perfectly separates the two classes of plants. This explains the high accuracy: **every point in the dataset is correctly classified**.

The network had both enough capacity and enough time to adjust its parameters.

But let's say we don't have time for 1000 training iterations. What happens if we reduce the number of training iterations to 400?

Let's find out:

```
parameters, losses = neural_network(X, y, n1=32, iterations=400, learning_rate
    =0.1)
```

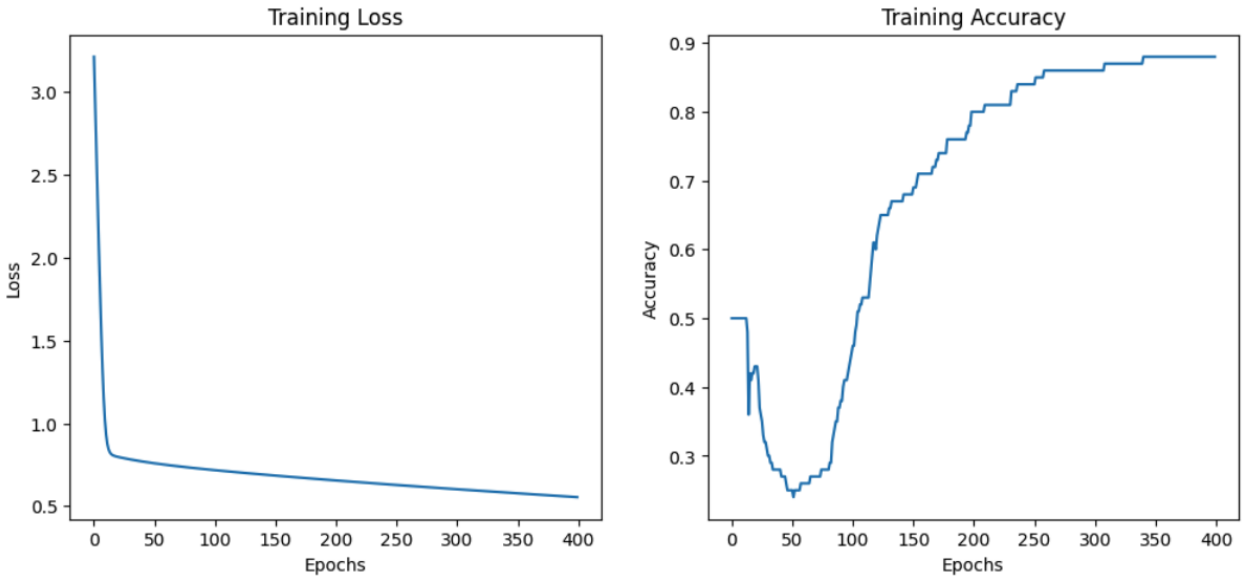



Figure 25: Loss and Accuracy 32 Neurons in the Hidden Layer (epochs=400)

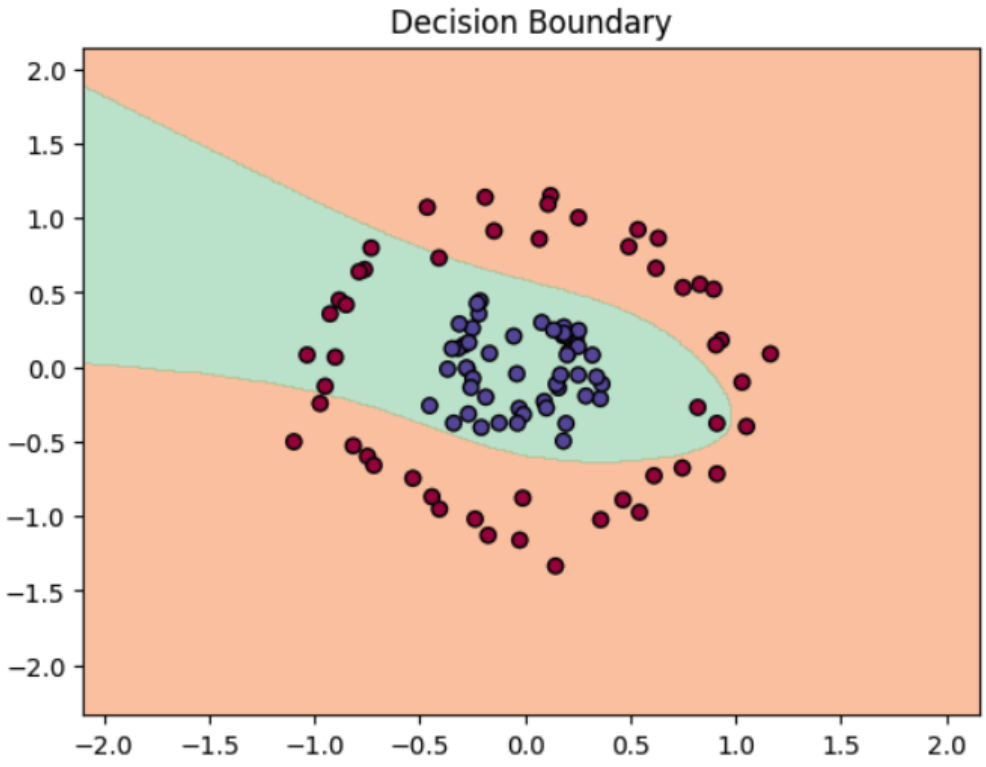



Figure 26: Decision Boundary with 32 Neurons in the Hidden Layer (epochs=400)

Even though the loss is still relatively low, we can see that:

- The decision boundary is not as sharp
- The network hasn't had enough time to fully adjust

One option would be to increase the learning rate to speed up convergence. But there's a risk: the model could overshoot the optimal point and **fail to converge**, oscillating instead of stabilizing.

However, we now have another powerful lever: **the number of neurons in the hidden layer**. Let's double it — from 32 to **64 neurons** — and see what happens to the model's expressiveness.

```
parameters, losses = neural_network(X, y, n1=64, iterations=400, learning_rate
    =0.1)
```

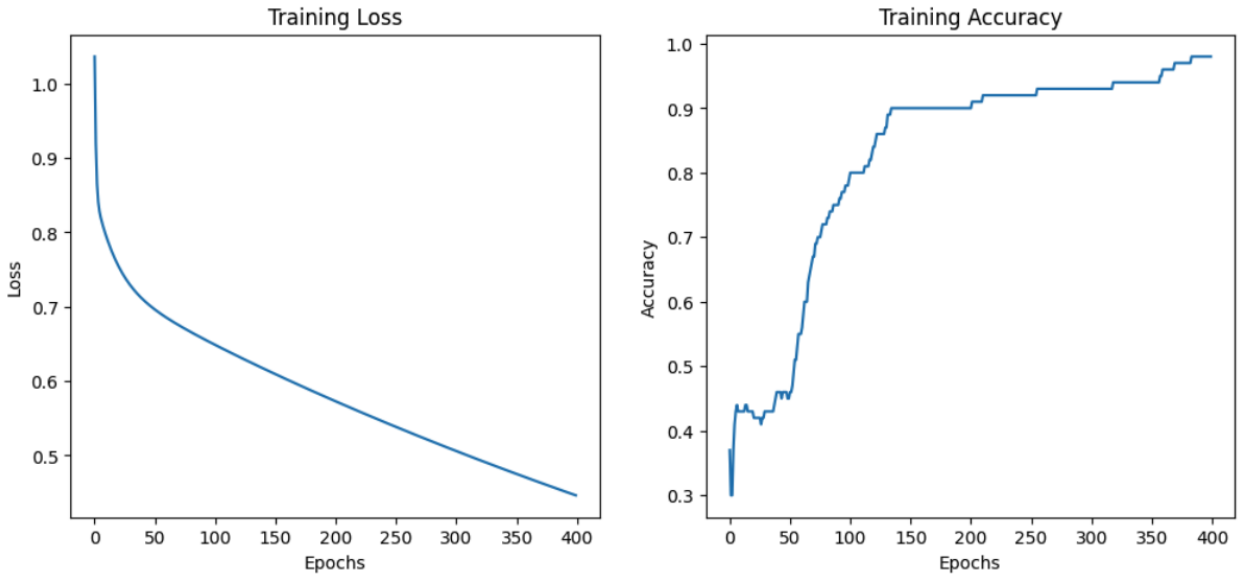



Figure 27: Loss and Accuracy 64 Neurons in the Hidden Layer (epochs=400)

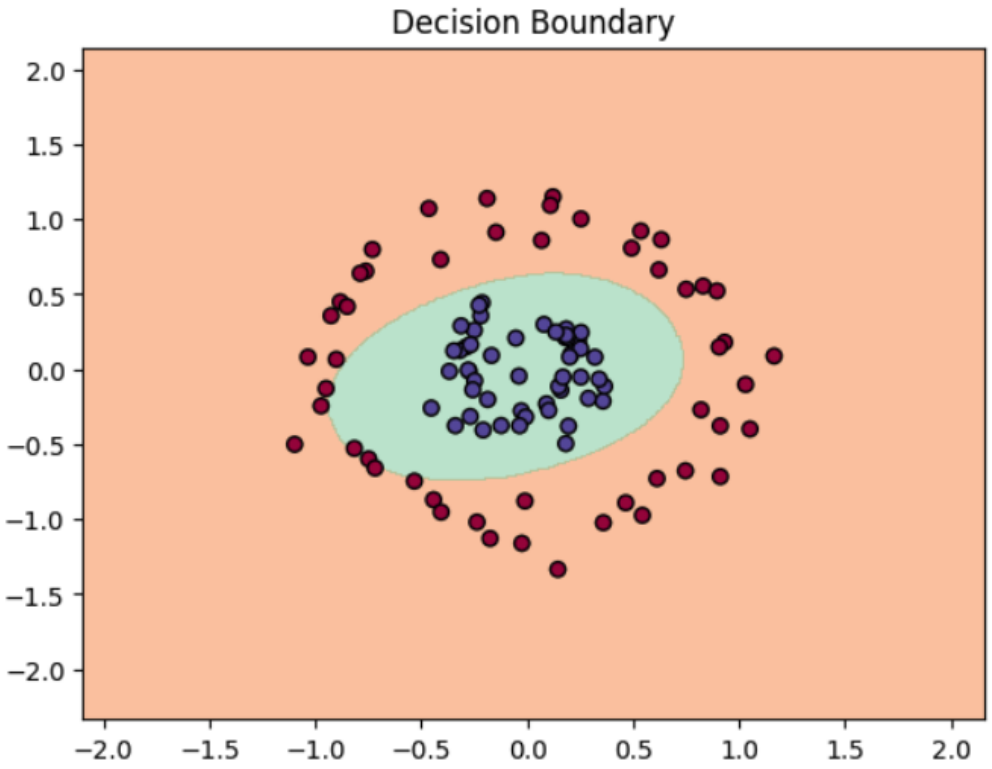


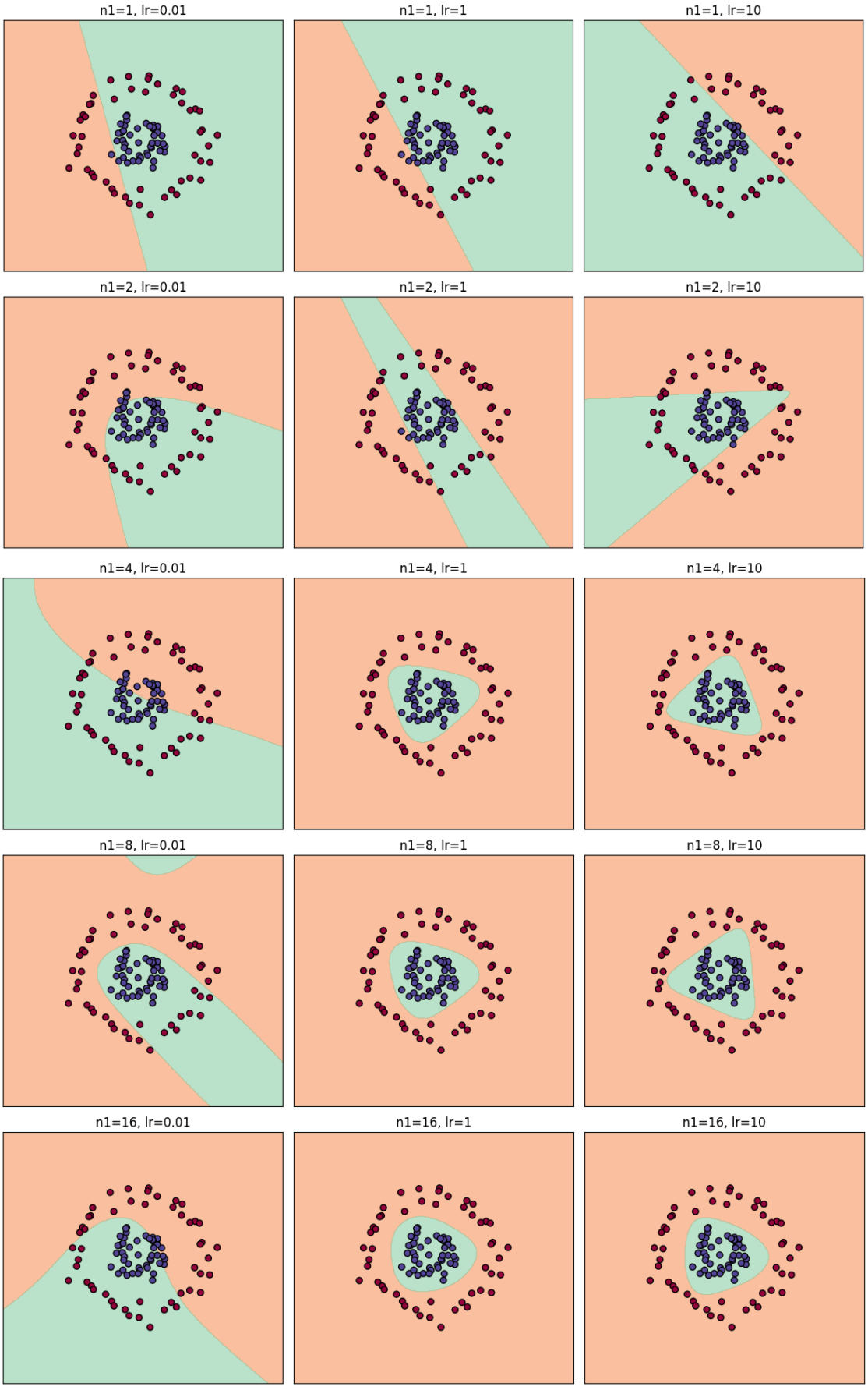

Figure 28: Decision Boundary with 64 Neurons in the Hidden Layer (epochs=400)

That worked. The accuracy is back to 100% — or very close to it.

We omit the code for successive experiments here for clarity. If you're curious, feel free to check the full implementation on GitHub or Google Colab.

Decision Boundary with 1, 2, 4, 8 and 16 neurons in the hidden layer and different learning rates. We fixed the number of epochs to 1000 for all experiments.

But here's the output of it all:


n1=1, lr=0.01
n1=1, lr=1
n1=1, lr=10
n1=2, lr=0.01
n1=2, lr=1
n1=2, lr=10
n1=4, lr=0.01
n1=4, lr=1
n1=4, lr=10
n1=8, lr=0.01
n1=8, lr=1
n1=8, lr=10
n1=16, lr=0.01
n1=16, lr=1
n1=16, lr=10

## Observations

### Number of Neurons in the Hidden Layer

- **1 neuron:**
  The model behaves like simple logistic regression — the decision boundary is linear.
  This is expected: one neuron with a sigmoid activation is essentially a linear classifier.

- **2 neurons:**
  The boundary becomes slightly curved (e.g., quadratic), but still lacks the expressiveness needed to separate the two classes.
  The model starts capturing non-linearities — but not enough.

- **4 neurons and above:**
  The decision boundary becomes noticeably more complex and flexible.
  The network now has enough capacity to separate the two classes correctly.

- *In short:*
  **More neurons → more expressive power** → better ability to learn non-linear patterns.

### Effect of Learning Rate (with fixed number of epochs)

- **Too small:**
  The model learns very slowly — it may not converge in time, or at all within the given number of epochs.

- **Too large:**
  The model overshoots the optimal point, leading to oscillations or even divergence of the loss.

  **Takeaway:** The learning rate must be tuned carefully — in harmony with the number of epochs and the complexity of your dataset.

## Why Is the Number of Neurons Often a Multiple of 2?

There is no theoretical constraint forcing you to use an even number of neurons in a hidden layer. However, in practice, multiples of 2 are often chosen — and for good reason:

- *Efficiency:*
  Vectorized operations are typically faster when tensor sizes are powers of 2 (e.g., 8, 16, 32, 64) — especially on GPUs.

- *Convention:*
  These values are widely used in tutorials, frameworks, and benchmarks — making them a natural starting point.

- *Reproducibility:*
  Choosing common sizes helps compare results and share code more easily across projects.

So while nothing stops you from using 3, 5 or 11 neurons — **starting with powers of 2 is often practical and efficient.**

## You Did It.

You now know how to:

- Initialize a neural network from scratch
- Perform forward propagation to compute predictions
- Compute the loss using log-loss
- Backpropagate gradients to optimize the model
- Monitor training with loss and accuracy
- Visualize the decision boundary
- Understand the effects of hyperparameters like learning rate and layer width

  You've built and trained a fully functional 2-layer neural network — and understood each step of the process.

This is the foundation of deep learning.
From here, you can scale, extend, and explore further.

**Ready for more?** Let's go deeper into architectures, regularization, and real-world datasets.

# Cat vs Dog Classification

Remember the *cat vs dog* dataset?

You now have everything you need to train a neural network that can classify images of cats and dogs.

**Let's put our neural network to the test.**

But before jumping into training — Let's ask the fundamental question:

## What's the most important thing to do before training a neural network?

Answer: **Data preprocessing.**

## Data Preprocessing

To work with images, we need to prepare the data carefully.

Here's what we need to do:

- **Load the data**: Read the cat vs dog dataset
- **Resize the images**: Set all images to the same shape (e.g. $64 \times 64$)
- **Normalize the pixel values**: Scale values between 0 and 1
- **Split the data**: Separate into training and test sets

```
X_train, y_train, X_test, y_test = load_data()
```

Let's visualize a few training examples:

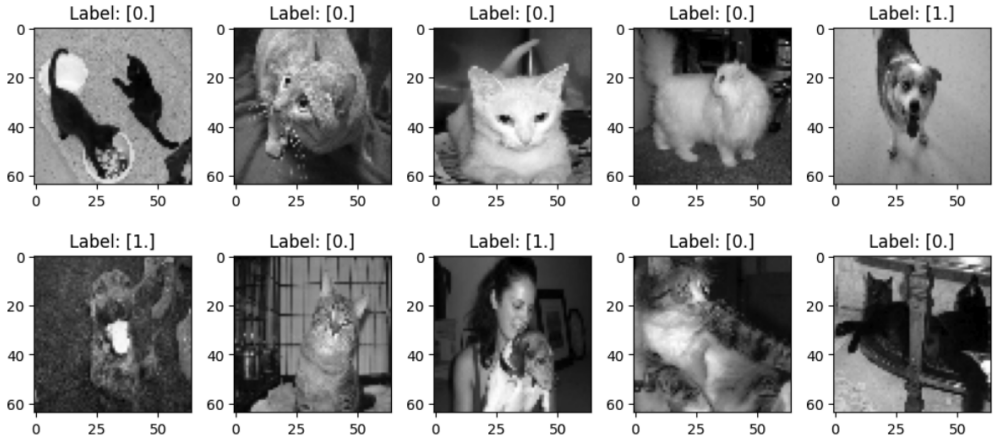



Figure 29: Examples from the Cat vs Dog Dataset

Since we're building a neural network, we want our input data to have the shape:

## Understanding the Shapes Step-by-Step

Let's take a closer look at how we prepare the dataset — and why we do it this way.

### Raw Dataset Shapes

```
X_train.shape, y_train.shape, X_test.shape, y_test.shape
# ((1000, 64, 64), (1000, 1), (200, 64, 64), (200, 1))
```

At this stage:

- Each image is of size $64 \times 64$ pixels
- We have:
  - 1000 images for training
  - 200 images for testing
- So `X_train` has shape $(1000, 64, 64)$: one $64 \times 64$ image per training sample
- `y_train` and `y_test` are column vectors of labels (0 = cat, 1 = dog)

### Transposing the Data

```
X_train = X_train.T
X_test = X_test.T
X_train.shape, X_test.shape
# ((64, 64, 1000), (64, 64, 200))
```

Why transpose?

> Because we want each image to become a **column vector** in our dataset.

So instead of:

`(num_samples, height, width)`

we want:

`(height, width, num_samples)`

This sets us up to flatten each image and stack them horizontally.

### Reshaping and Normalizing

```
X_train_reshaped = X_train.reshape(-1, X_train.shape[-1]) / X_train.max()
X_test_reshaped = X_test.reshape(-1, X_test.shape[-1]) / X_test.max()
X_train_reshaped.shape, X_test_reshaped.shape
# ((4096, 1000), (4096, 200))
```

Each $64 \times 64$ image now becomes a vector of 4096 features (pixels). All pixel values are scaled between 0 and 1 by dividing by the maximum value (255).

`X_train_reshaped` has shape:

$$(4096, 1000) \quad \rightarrow \quad \text{4096 input features for 1000 training examples}$$

### Labels

```
y_train.shape, y_test.shape
# ((1000, 1), (200, 1))
```

No transformation is needed here.
Each label remains a scalar (0 or 1), stored as a column vector.

With this format:

- The input $X$ can be fed directly into a neural network
- The label $y$ matches the expected output shape

Your dataset is now clean, normalized, and correctly shaped.

**From raw images to input matrices — your dataset is ready.**

## Training on Cat vs Dog

Now that our dataset is preprocessed and ready, we can train our neural network on real image data.

We'll use the same architecture as before:

- One hidden layer
- Sigmoid activation
- Binary output

Let's start with 32 neurons in the hidden layer, a learning rate of 0.1, and train for 1000 epochs.

```
parameters, losses, accuracies = neural_network(
    X_train_reshaped, y_train,
    n1=32, iterations=1000,
    learning_rate=0.1
)
```

## Evaluating the Model on Test Data

To check how well the model generalizes, we perform a forward pass on the test set:

```
activations_test = forward_propagation(X_test_reshaped, parameters)
A_test = activations_test['A2']

from sklearn.metrics import accuracy_score, log_loss

test_preds = (A_test > 0.5)
test_loss = log_loss(y_test.flatten(), A_test.flatten())
test_accuracy = accuracy_score(y_test.flatten(), test_preds.flatten())

print(f"Test Loss: {test_loss:.4f}")
print(f"Test Accuracy: {test_accuracy:.2%}")
```

## Results and Interpretation


Figure 30: Training and Test Loss/Accuracy and the Train/Test dataset

### What's Happening Here?

Our model is clearly **overfitting**.

- The training loss drops, and accuracy goes up to 100%
- But the test loss rises again, and test accuracy stays low

*It learns the training set — but can't generalize.*

Maybe the model lacks intelligence ? Maybe the architecture is too simple?

Let's not hold back.

What if we gave the model more freedom? More neurons. More layers. More depth.

What happens when we give the network the full capacity to learn?

Let's go beyond our two-layer architecture and build a **fully connected neural network** — one where we can choose:

- As many hidden layers as we want
- As many neurons per layer as we need

This type of model can, in theory, approximate any function — and will give us much more expressive power.

This is the final step in our journey. Coding a brain from scratch. See you beyond the last wall, at `practice_03.ipynb`.

practice_03.ipynb

**Launch practice_03.ipynb on Google Colab**

*Follow this part on Google Colab here.*
*Code and Train your own deep neural network from scratch in this pre-coded space.*
*You'll realize it's not that deep.*

# Hang in there, my friend - this is the turning point.

Today, we are coding this from scratch:

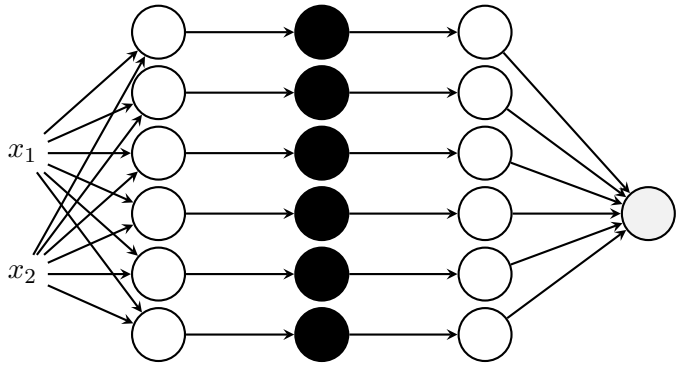



Figure 31: Fully connected network with infinitely many hidden layers (conceptual representation).

# From Diagram to Code: Artificial Neural Network.

You've now seen what a fully connected neural network looks like — from inputs to outputs, through multiple layers of neurons. Let's now build it from scratch in code.

Our goal is simple:

> **Create a neural network where we can define any number of layers and neurons.**

No black boxes, no shortcuts. Just Python, NumPy, and clean mathematics.

### Architecture Definition

Before training a neural network, we need to define its architecture.

We need to answer two fundamental questions:

- How many layers?

- How many neurons in each layer?

To address this, we will create a function that initializes the parameters of our network. This function will encode both the number of layers and the number of neurons in each layer.

Each layer $l$ will be associated with:

- a weight matrix $\mathbf{W}^{[l]}$
- a bias vector $\mathbf{b}^{[l]}$

These matrices are not arbitrary. Their dimensions tell us precisely how many neurons are present in each layer — and how each layer is connected to the previous one.

*So how do we encode this architecture in code?*

With a simple list.

> The `length` of the list tells us how many layers there are.
> The `value at each index` tells us how many neurons are in that layer.

A simple and expressive design.
Let's now translate it into Python.

Here is the initialization function:

```python
def initialisation(dimensions):
    parameters = {}
    L = len(dimensions)

    for l in range(1, L):
        parameters[f"W{l}"] = np.random.randn(dimensions[l], dimensions[l - 1])
        parameters[f"b{l}"] = np.random.randn(dimensions[l], 1)

    return parameters
```

Listing 1: Initialisation of parameters

Note how the weights and biases are initialized randomly for each layer. But most importantly: how the `dimensions` list governs the shape of each weight and bias matrix.
The matrix $\mathbf{W}^{[l]}$ has shape (neurons in layer $l$, neurons in layer $l-1$), and the bias vector $\mathbf{b}^{[l]}$ has shape (neurons in layer $l$, 1).

We now define the shape of our neural network and use this function to initialize all parameters:

```python
dimensions = [2, 32, 32, 1]
parameters = initialisation(dimensions)

for key, value in parameters.items():
    print(f"{key} {value.shape}")
```

This produces the following output:

```
W1 (32, 2)
b1 (32, 1)
W2 (32, 32)
b2 (32, 1)
W3 (1, 32)
b3 (1, 1)
```

This output tells us:

- The first layer has 32 neurons, each connected to the 2 input features.
- The second layer has 32 neurons, each connected to the previous layer's 32 neurons.
- The output layer has 1 neuron, connected to the last hidden layer's 32 neurons.

You have just defined a complete fully connected neural network.

# Next step: Give it a purpose.

### Forward Propagation

Let's now bring this architecture to life.

Recall from the previous notebook: Each layer takes as input the output of the previous one. The input layer receives the raw data $X$, while each subsequent layer receives the activation of the previous layer.

We now generalize this behavior.

> The $i$-th layer takes as input the activation $A^{[i-1]}$ from the previous layer, applies a linear transformation, and then passes it through a non-linear activation function.

At each layer $l$, we compute:

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}, \quad A^{[l]} = \sigma(Z^{[l]})$$

This creates a chain-like structure where information flows from layer to layer, gradually transforming raw input into high-level features.

Here is the function that implements this process:

```python
def forward_propagation(X, parameters):
    activations = {}
    L = len(parameters) // 2  # total number of layers

    A = X
    activations['A0'] = A
```

```
    for l in range(1, L + 1):
        W = parameters[f'W{l}']
        b = parameters[f'b{l}']

        Z = np.dot(W, A) + b
        A = 1 / (1 + np.exp(-Z))  # sigmoid activation

        activations[f'A{l}'] = A

    return activations
```

Listing 2: Forward propagation through all layers

We now run this function on our input data:

```
activations = forward_propagation(X, parameters)

for key, value in activations.items():
    print(key, value.shape)
```

This produces the following output:

```
A0 (2, 100)
A1 (32, 100)
A2 (32, 100)
A3 (1, 100)
```

Interpretation:

- The input $X$ contains 100 examples with 2 features — hence $A^{[0]}$ has shape $(2, 100)$.
- The first and second hidden layers each have 32 neurons, and compute activations over all 100 examples — hence $(32, 100)$.
- The output layer has a single neuron — one prediction per example.

# From raw data to prediction — layer by layer.

## Backpropagation

Once the network computes a prediction, we need to measure how wrong it is — and more importantly, how to correct it. This is the role of **backpropagation**.

We start from the output layer and work backward, computing the gradients of the loss with respect to each parameter in the network.

Let's begin with a concrete example: a network with two layers.

$$\frac{\partial \mathcal{L}}{\partial W_2} = \frac{1}{m}(A_2 - Y) \cdot A_1^\top$$
$$\frac{\partial \mathcal{L}}{\partial b_2} = \frac{1}{m}\sum_{i=1}^{m}(A_2 - Y)^{(i)}$$
$$\frac{\partial \mathcal{L}}{\partial W_1} = \frac{1}{m} dZ_1 \cdot X^\top$$
$$\frac{\partial \mathcal{L}}{\partial b_1} = \frac{1}{m}\sum_{i=1}^{m} dZ_1^{(i)}$$

with:

$$dZ_2 = A_2 - Y$$
$$dZ_1 = (W_2^\top \cdot dZ_2) \circ A_1 \circ (1 - A_1)$$

**Generalization to any layer $l$**

Take a moment to observe the pattern. Each layer computes its local error $dZ_l$ based on the error from the layer above and the derivative of its activation function.

So for the last layer:

$$dZ_L = A_L - Y$$

and for all hidden layers $l < L$:

$$dZ_l = (W_{l+1}^\top \cdot dZ_{l+1}) \circ A_l \circ (1 - A_l)$$

That's why, for any layer $l$ in a network of $L$ layers, we have:

$$\frac{\partial \mathcal{L}}{\partial W_l} = \frac{1}{m} dZ_l \cdot A_{l-1}^\top$$
$$\frac{\partial \mathcal{L}}{\partial b_l} = \frac{1}{m}\sum_{i=1}^{m} dZ_l^{(i)}$$

where:

- $m$ is the number of training examples,
- $A_{l-1}$ is the activation from the previous layer,
- $dZ_l$ is the local error at layer $l$,
- $\circ$ denotes the element-wise (Hadamard) product.

**Backpropagation works by:**

1. Computing the output error $dZ_L$
2. Propagating it backward through the network using the weights and the sigmoid derivative
3. Computing gradients $\frac{\partial \mathcal{L}}{\partial W_l}$ and $\frac{\partial \mathcal{L}}{\partial b_l}$ at each layer
4. Using these gradients to update the parameters via gradient descent

Let's now translate this logic into code:

```python
def back_propagation(y, parameters, activations):
    m = y.shape[1]
    L = len(parameters) // 2
    dZ = activations[f'A{L}'] - y
    gradients = {}

    for l in reversed(range(1, L + 1)):
        A_prev = activations[f'A{l-1}']
        W = parameters[f'W{l}']

        gradients[f'dW{l}'] = (1/m) * np.dot(dZ, A_prev.T)
        gradients[f'db{l}'] = (1/m) * np.sum(dZ, axis=1, keepdims=True)

        if l > 1:
            dA_prev = np.dot(W.T, dZ)
            dZ = dA_prev * A_prev * (1 - A_prev)

    return gradients
```

Listing 3: General backpropagation algorithm

Let's unpack what just happened.

This function may look concise — but it captures the full elegance of the backpropagation algorithm.

We begin at the output layer: $dZ = A_L - Y$ — the raw difference between prediction and ground truth. From there, we iterate backwards. At each layer, we compute the gradient of the loss with respect to the weights and biases using clean, vectorized expressions:

$$\frac{\partial \mathcal{L}}{\partial W_l} = \frac{1}{m} dZ_l \cdot A_{l-1}^\top, \quad \frac{\partial \mathcal{L}}{\partial b_l} = \frac{1}{m} \sum dZ_l$$

Then comes the key insight: the chain rule. We take the upstream gradient and flow it backward through the weights:

$$dA_{l-1} = W_l^\top \cdot dZ_l$$

We then apply the derivative of the sigmoid function:

$$dZ_{l-1} = dA_{l-1} \circ A_{l-1} \circ (1 - A_{l-1})$$

This gives us the local error for the previous layer — and the cycle continues.

In just a few lines, we've implemented a fully general backpropagation engine — one that can scale to any depth, and prepare the ground for efficient learning via gradient descent.

*From abstract math to concrete code — this is the heart of learning.*

We now run the function and inspect the shape of each computed gradient:

```python
gradients = back_propagation(y, parameters, activations)
for key, value in gradients.items():
    print(key, value.shape)
```

This produces the following output:

```
dW3 (1, 32)
db3 (1, 1)
dW2 (32, 32)
db2 (32, 1)
dW1 (32, 2)
db1 (32, 1)
```

Each gradient is shaped to exactly match the parameter it updates:

- **Layer 3 (Output)**
  `dW3 (1, 32)` connects 32 activations from the last hidden layer to 1 output neuron
  `db3 (1, 1)` is the bias for the output neuron

- **Layer 2 (Hidden)**
  `dW2 (32, 32)` connects 32 neurons from layer 1 to 32 neurons in layer 2
  `db2 (32, 1)` is the bias vector for each neuron in layer 2

- **Layer 1 (Hidden)**
  `dW1 (32, 2)` maps the input features (2-dimensional) to the first hidden layer (32 neurons)
  `db1 (32, 1)` is the bias vector for layer 1

This layout mirrors the structure of the network itself — from output to input. Each gradient flows in reverse order, ready to update the corresponding parameter in a single pass.

This is how learning happens: through structure, through precision, and through alignment between theory and implementation.

This is more than just a shape check — it's a validation of the entire learning pipeline.

Every derivative, every matrix product, every layer — it all fits. The math and the code speak the same language.

Each gradient is now ready to be used for one purpose: to make the model better.

# Backpropagation demystified. Time to use it.

## Gradient Descent: Updating the Parameters

Backpropagation gave us the gradients — but gradients alone don't change anything.

To improve the model, we need to update the parameters in the direction that reduces the loss. That's what gradient descent does.

At each layer $l$, we perform the update:

$$W^{[l]} \leftarrow W^{[l]} - \alpha \cdot \frac{\partial \mathcal{L}}{\partial W^{[l]}}$$
$$b^{[l]} \leftarrow b^{[l]} - \alpha \cdot \frac{\partial \mathcal{L}}{\partial b^{[l]}}$$

Where $\alpha$ is the learning rate — a small scalar that controls the step size of the update.

Here is the Python function that applies this rule to every layer:

```python
def update_parameters(parameters, gradients, learning_rate):
    L = len(parameters) // 2

    for l in range(1, L + 1):
        parameters[f'W{l}'] -= learning_rate * gradients[f'dW{l}']
        parameters[f'b{l}'] -= learning_rate * gradients[f'db{l}']

    return parameters
```

Listing 4: Parameter update using gradient descent

This simple loop performs one update step across all layers.

- `parameters` holds the current weights and biases
- `gradients` comes from backpropagation
- `learning_rate` determines how fast (or slow) the model learns

Each parameter takes a small step in the direction that reduces the loss — bringing the network closer to a better set of predictions.

# This is where the network becomes alive.

## Training Loop

We now have all the pieces.

- **Forward propagation** to make a prediction
- **Backpropagation** to compute the error and how to correct it
- **Gradient descent** to apply that correction

Individually, they are just mechanisms. Together — they become something more.

**It becomes the neural network.**

And we train it using the same **training loop** that contains three steps:

1. It receives input and computes an output.
2. It evaluates how wrong it was.
3. It rewires itself — slightly — to do better next time.

Iteration after iteration, this cycle repeats. The weights shift. The loss decreases. The predictions improve.

Here is the code that brings the network to life:

```
def neural_network(X, y, dimensions, learning_rate=0.1, iterations=1000):
    parameters = initialisation(dimensions)

    for i in range(iterations):
        # Forward pass
        activations = forward_propagation(X, parameters)

        # Backward pass
        gradients = back_propagation(y, parameters, activations)

        # Parameter update
        parameters = update_parameters(parameters, gradients,
    learning_rate)

        # Optional: track the loss
        if i % 100 == 0:
            A_final = activations[f'A{len(dimensions) - 1}']
            loss = -np.mean(y * np.log(A_final) + (1 - y) * np.log(1 -
    A_final))
            print(f"Iteration {i}: loss = {loss:.4f}")

    return parameters
```

Listing 5: The training loop — where learning happens

This is not just a loop.

It is a feedback system. An adaptive structure. A recursive engine that takes raw input and sculpts its own internal logic to match the world.

You can control how fast it learns (`learning_rate`), how long it trains (`iterations`), or how deep and wide it thinks (`dimensions`).

But once launched, this function learns — by itself.

## From Prediction to Decision: Toxic or Not?

Let's now return to our original goal: *classify whether a plant is toxic or not, based on its features.*

To explore this, we will simulate a variety of synthetic datasets that represent labelled toxic (class 1) and non-toxic (class 0) plants. For each one, we'll let the neural network classify the inputs — and visualize the decision boundary it creates.

> This is how we see what the network "thinks".
> How it splits space. Where it draws the line. Literally.

Here's how we'll do it:

1. Generate a dataset (e.g. moons, circles, blobs, spirals)
2. Train the neural network using our `neural_network()` function
3. Visualize the final decision surface: which regions are predicted toxic ($\hat{y} = 1$) and which are not ($\hat{y} = 0$)

The prediction itself is simple:

```
def predict(X, parameters):
    activations = forward_propagation(X, parameters)
    A_final = activations[f'A{len(parameters) // 2}']
    return (A_final >= 0.5)
```

Listing 6: Predicting class from probability

This function gives a final binary decision for each plant, based on the output activation. But what's far more interesting — is the structure the network built to make that decision.

Let's now visualize that internal geometry. We'll paint the input space according to the model's belief: toxic or not toxic. And then we'll overlay the real data to see what the model understood.

# The boundary is the belief. Let's see how it thinks.

Let's try it on different datasets and see how the neural network adapts.

## First Dataset: Random Blobs

We generate a synthetic binary classification problem using two clusters of points.

```
X, y = make_blobs(n_samples=200, centers=2, random_state=42, cluster_std=1.5)
```

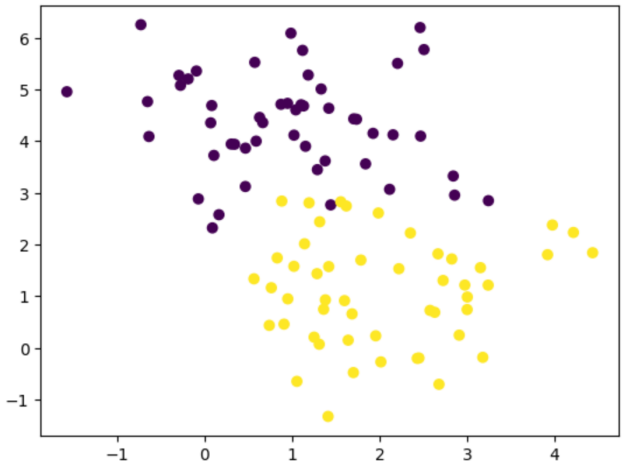


Figure 32: Random Dataset — Leaf Width vs. Length

We chose a two-hidden-layer neural network with 64 neurons in each layer:

```
dimensions = [2, 64, 64, 1]
```

And we train it with a learning rate of 0.1 for 500 epochs:

```
neural_network(X, y, [2, 64, 64, 1], learning_rate=0.1, epoch=500)
```

Training progression:

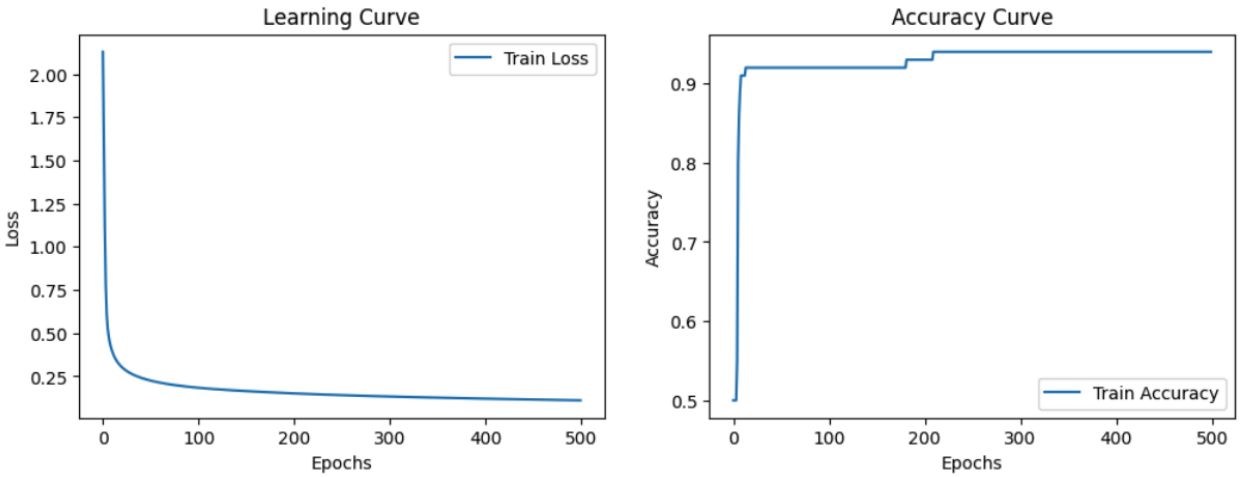



Figure 33: Training Progress on Random Dataset

And now, the decision boundary:

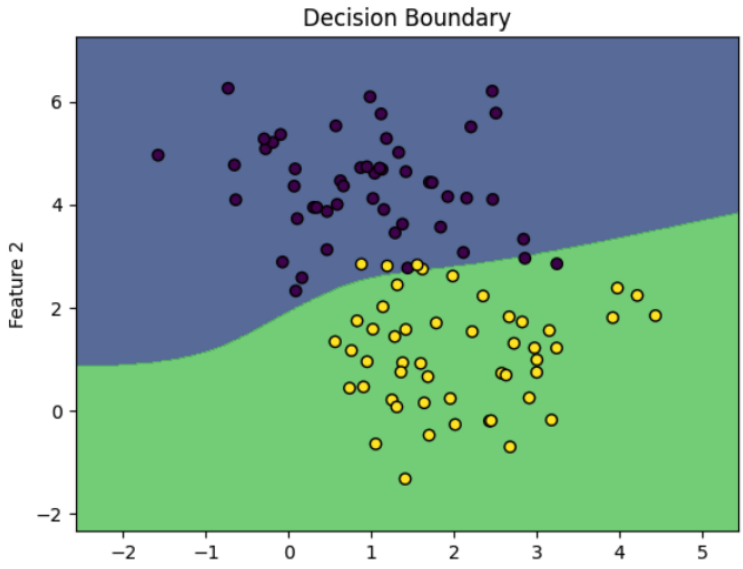



Figure 34: Decision Boundary for Random Dataset

## Second Dataset: Concentric Circles

This one's harder. Two intertwined classes. Let's see if the network can handle it.

```
X, y = make_circles(n_samples=100, noise=0.1, factor=0.3, random_state=0)
neural_network(X, y, [2, 64, 64, 1], learning_rate=0.1, epoch=500)
```

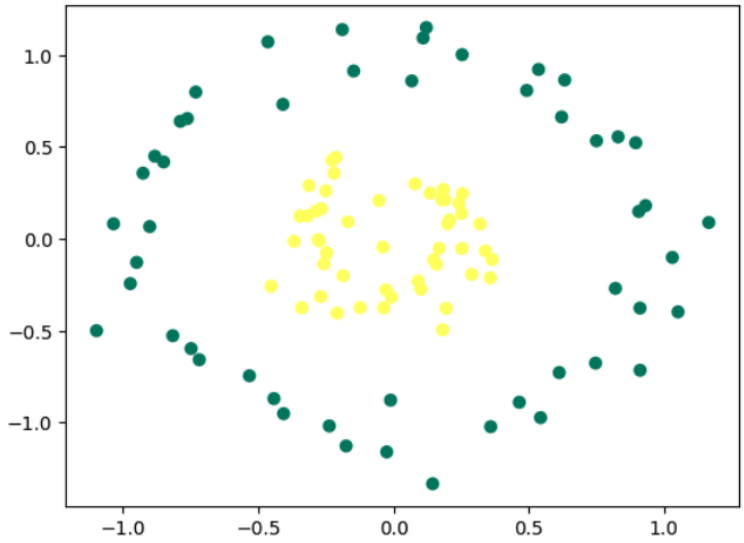



Figure 35: Concentric Circles Dataset

Longer training — more complexity:

```
neural_network(X, y, [2, 64, 64, 1], learning_rate=0.1, epoch=5000)
```

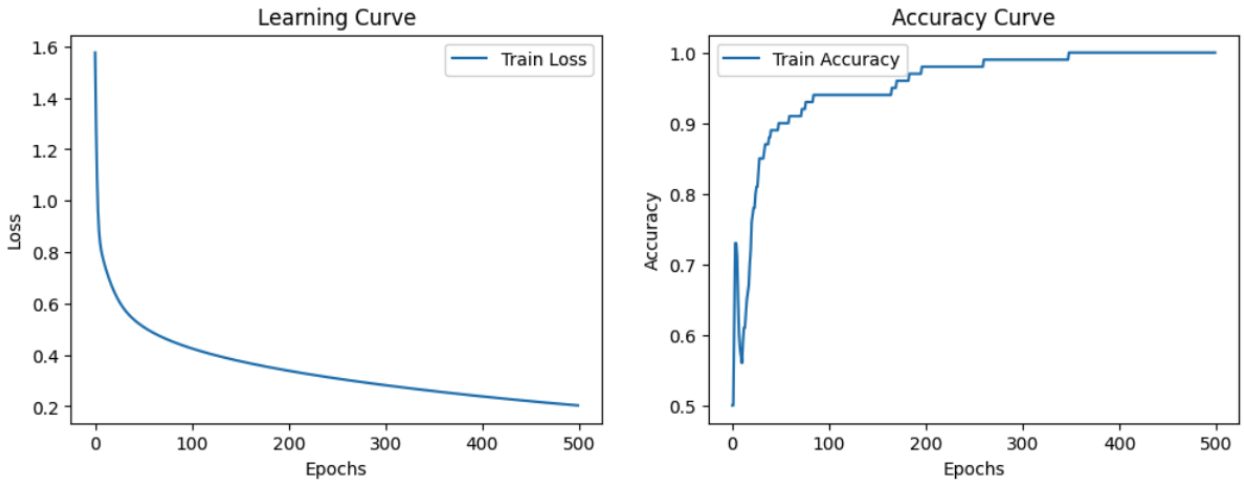



Figure 36: Training Progress on Circles Dataset

Final decision boundary:

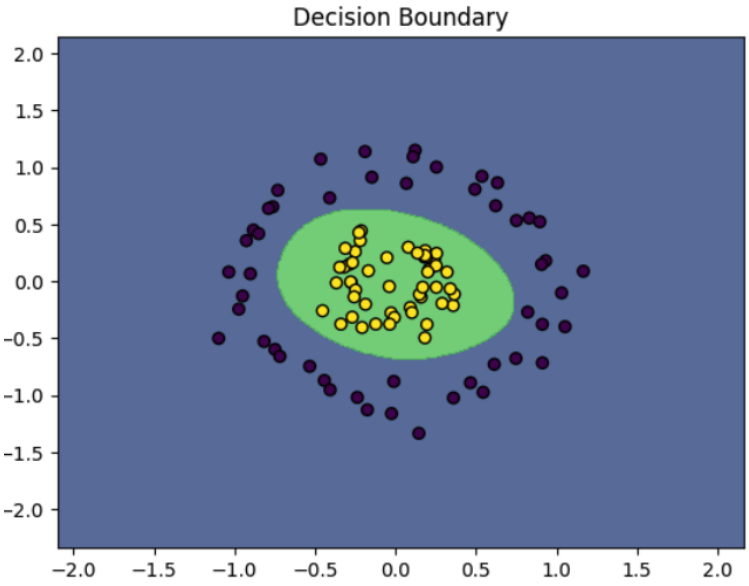



Figure 37: Decision Boundary for Circles Dataset

*Now you understand the power of a neural network ?*

**EVERY dataset can be classified now.**

You now have an independant learning tool that classifies nearly any dataset.

`And you built it from scratch.`

## Can this Neural Network understands Moons dataset?

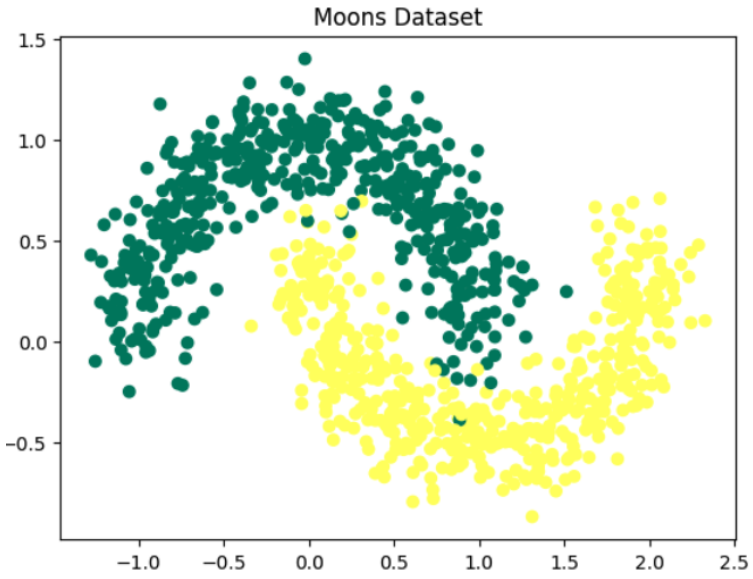



Figure 38: Moons Dataset

```
neural_network(X, y, [2, 64, 64, 1], learning_rate=0.1, epoch=5000)
```

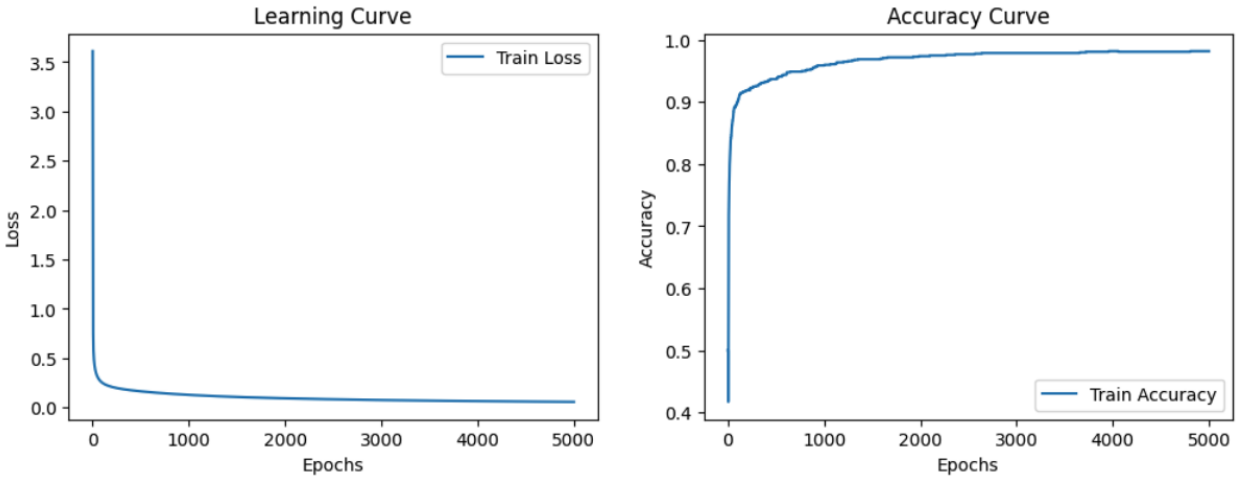



Figure 39: Training Progress on Moons Dataset

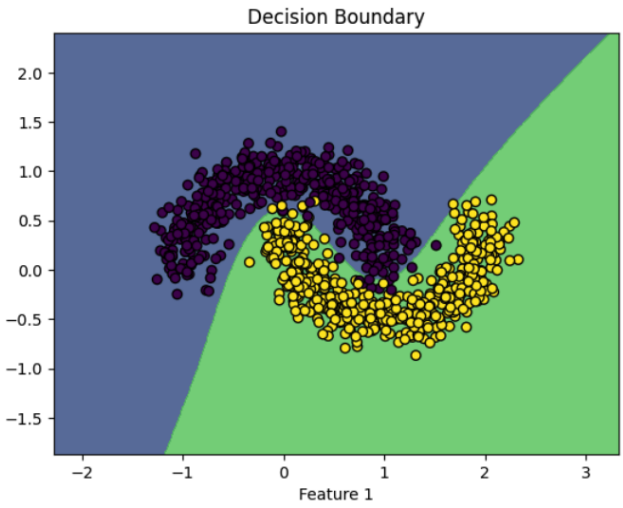



Figure 40: Decision Boundary for Moons Dataset

Okay, I know what you're thinking: *"This is just a toy example. It's too easy."*

## Fourth Dataset: Spirals

Let's now create a much more challenging dataset. Not linearly separable. Not even close. Spirals.

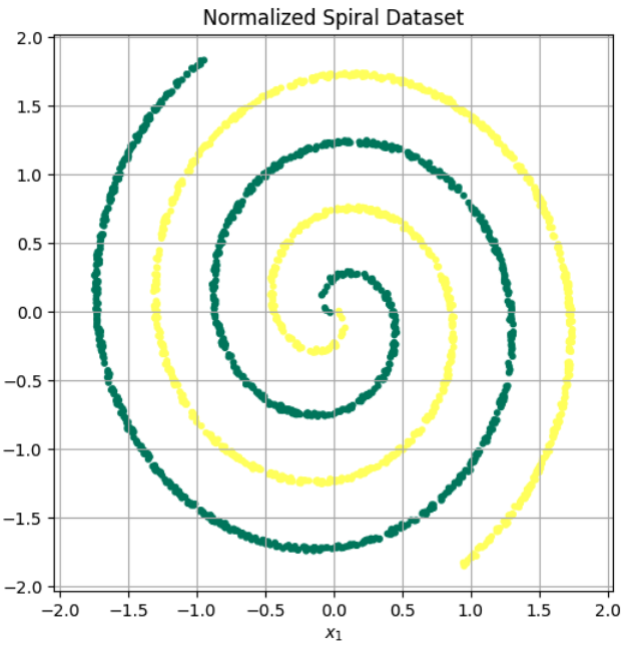



Figure 41: Spiral Dataset

We train the same 2-layer network on this data:

```
neural_network(X, y, [2, 64, 64, 1], learning_rate=0.1, epoch=5000)
```

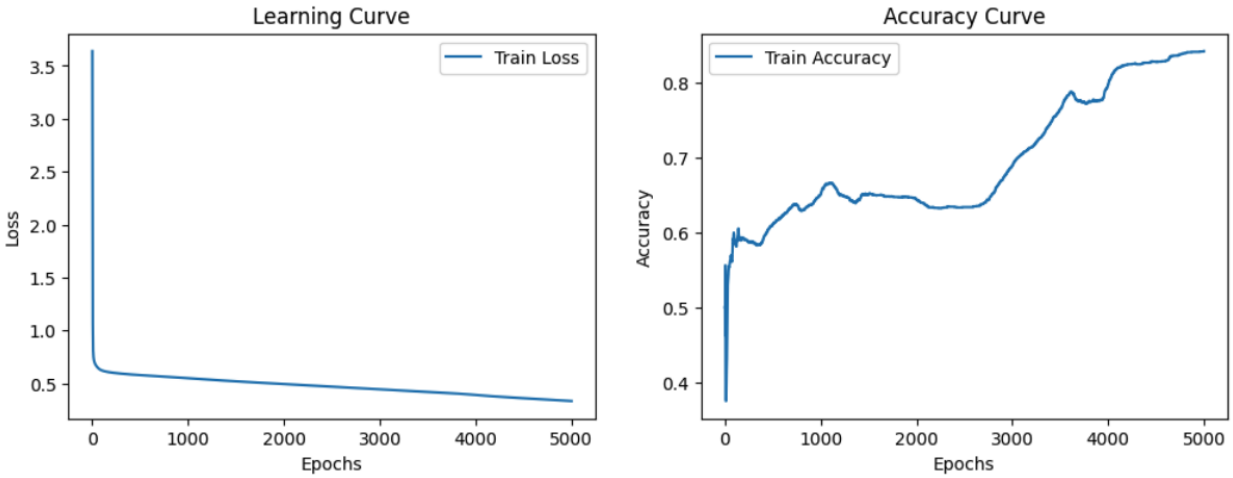



Figure 42: Learning Curves

The model tries, but... *The accuracy remains low. Something's off.*

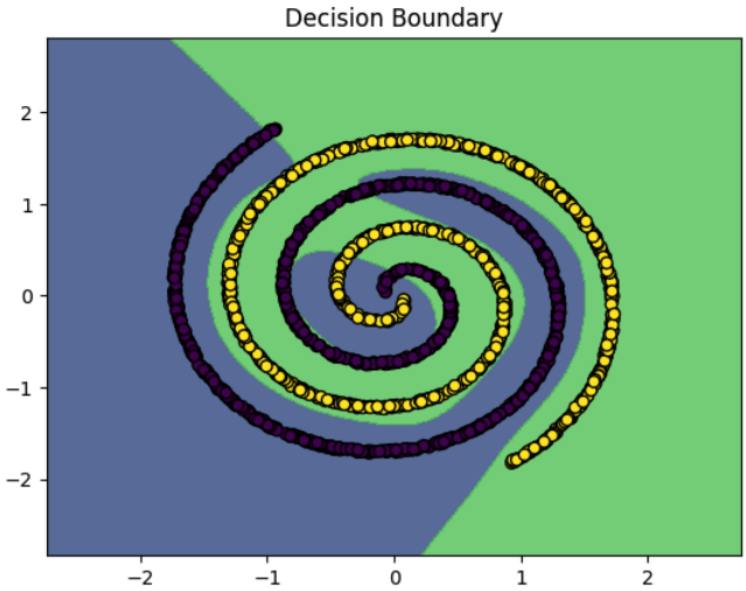



Figure 43: Decision Boundary for Spiral Dataset – 2 Layers

We now face a limitation. The network can't separate the spirals.

What can we do?

- Increase the number of **epochs**? Maybe.
- Increase the **learning rate**? Risky.
- **Add more neurons, more layers**? Now we're talking.

## Upgrading the Brain

Let's go from: `[2, 64, 64, 1]` to `[2, 128, 128, 128, 128, 1]`. That's 4 hidden layers with 128 neurons each.

```
neural_network(X, y, [2, 128, 128, 128, 128, 1], learning_rate=0.1, epoch=5000)
```

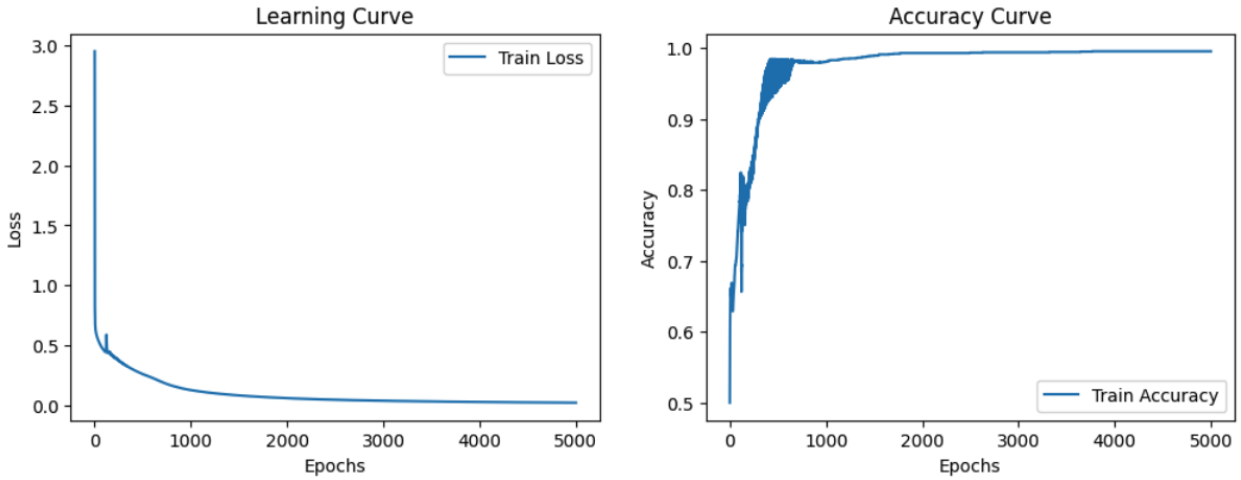



Figure 44: Learning Curves for Spiral Dataset – 4 Layers

And now, the decision boundary:

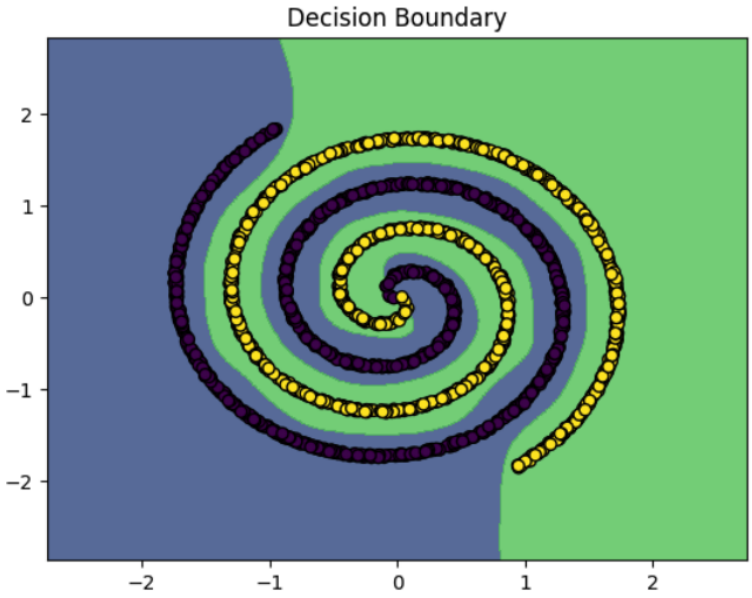



Figure 45: Decision Boundary for Spiral Dataset – 4 Layers

*Oh yeah, that worked.*
We just built a brain that understands spirals. But do you know what that really means ?

## A Neural Network: Our Creation

A neural network is not just a collection of equations. It is a structure we designed, a brain we shaped, a structure we designed, a logic we taught.

Let's take a moment to appreciate that.
We control its complexity. We choose how deeply it can think. We are, in a sense, its creators. Maybe he thinks we're gods. Maybe he's right. Maybe it's planning to create its own religion or something like that. For now, though...

**Let's keep using it to classify datasets.**

## What About Performance?

Sometimes — of course — the model is not perfect. But what matters is this: we can *see* the network searching, adjusting, learning. Even when the decision boundary isn't flawless, we witness the model thinking.

For example, let's say we reached a final training accuracy of 85%. That's a solid first step.

**But is 85% good enough?** That depends entirely on your objective.

- For a self-driving car? → Absolutely not.
- For a toy classification task? → Definitely yes.

But let's be cautious.

**Adding more layers and neurons doesn't always help.** It can increase the capacity of the network — yes — but there are some downsides of course, this world is balanced.

For example, it can also make the model memorize even the noise in your data instead of learning the underlying patterns.

This phenomenon is called **overfitting**.

Overfitting happens when a model performs very well on the training data, but poorly on unseen data.

*It's like a student who memorizes past exams instead of understanding the subject.*

That's why, as we increase the complexity of our model, we must also develop tools to measure generalization: validation data, regularization, early stopping, and more.

`But that's a story for other guides and topics.`

## Final Point on Overfitting

At this point, you might be wondering: *"What about the performance on the cat/dog dataset?"*

Let's see the final [1, 128, 128, 128, 128, 1] neural network on the cat/dog dataset we had :

```
neural_network(X, y, [2, 128, 128, 128, 128, 1], learning_rate=0.1, epoch=5000)
```

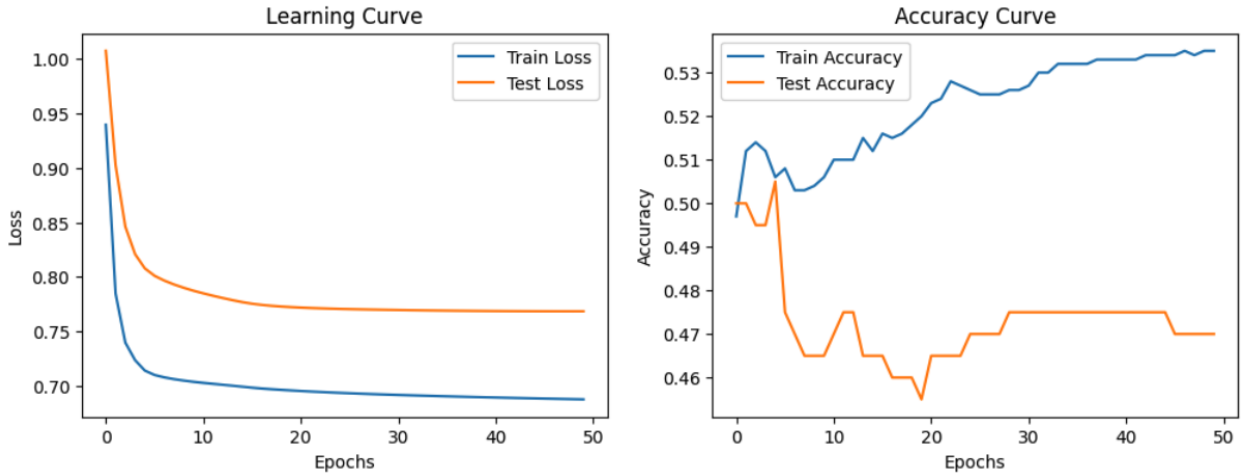



Figure 46: Training Progress on Cat/Dog Dataset

We still have an overfitting problem.

This is, in fact, a textbook case of **overfitting**. The neural network is simply too powerful for the data we have.

There is no trick here. **Every model will fail.**

But why?

We now face one of the most fundamental, million-dollar questions in machine learning:

**Is the problem in the data, the model, or the training process?**

Let's dissect this.

Suppose the model is well-designed. Still, we observe poor performance on the test set. Then, the culprit is likely the data itself.

What does it mean for data to be 'not good enough'?

- It might be too small.
- It might lack diversity.
- It might not capture the true complexity of the task.

The cat/dog dataset is a perfect illustration. It contains only **1,000 images**—far too few to train a high-capacity neural network without overfitting.

And that's precisely why this guide ends here.

Don't fall into the trap of thinking that every problem can be solved with a neural network. Don't believe that the solution lies hidden in some mathematical trick, in a piece of code, or in someone else's mind.

**The solution is in the process.**

In how we define the problem. In how we choose, structure, and question the data. In the model we design. In the way we train, test, and deploy. In every iteration, and every lesson along the way.

That's why a true data scientist is so valuable. Because they've walked the path—*without shortcuts*. Because they understand that the value is not just in the answer, but in the journey to reach it.

# Let's stop here this journey.

## Conclusion

**You can be proud of yourself.**
You've taken the lead in your own learning process. And that changes everything.

Now that I've lightened the candle, will you start the fire?

Take a moment. Reflect on what you've learned, the code you've written, the concepts you've absorbed. And then, look at this:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(64, activation='sigmoid', input_shape=(2,)),
    tf.keras.layers.Dense(64, activation='sigmoid'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.compile(optimizer='sgd', loss='binary_crossentropy', metrics=['
    accuracy'])
model.fit(X, y, epochs=500, batch_size=32)
```

Listing 7: Neural Network with TensorFlow

This is a neural network built with TensorFlow, the very same architecture we just implemented from scratch. In just a few lines of code, you see it all: the activation function (Sigmoid), the optimizer (Gradient Descent Algorithm), the architecture, the loss function ($\log_l oss$).$It'sallthere$.

But now, you understand what's behind it.
Not just the *how*. You've started understanding the *why*.

Because here's the truth:

**The point is not to use the best library.**
**The point is to understand the reason it exists.**

That's what makes a real data scientist. That's what makes someone able to solve actual problems — not just benchmark datasets. That's what makes you valuable. Rare. Expensive.

You now know how to train a neural network from scratch, how to visualize its decisions, how to make it learn. You've built the muscle memory of forward propagation, of backpropagation, of gradient descent.

And more importantly: you've built your own mental model of it.

But remember:

**This is just the beginning.**

Understanding the internal mechanics is one level. But understanding **when**, **how**, and **why** to use it — that's a whole new game.

That's what my next course will be about: not just learning to code, but understanding what it means to **be** a data scientist.

You don't need a degree. Not a bachelor's. Not a master's. Not an engineering school.

Sure, those paths offer structure — professors, exercises, projects, deadlines. But they don't offer the one thing that matters most:

**The decision to learn.**
**The discipline to keep going.**

And that's what you've done.

You chose this path. You decided to learn — not because you had to, but because something inside told you it mattered.

I'm not here to offer you comfort.
I'm not here to sugarcoat your journey.

I'm here because — like you — I chose to act.
I went through schools, degrees, certifications. But the things that shaped me the most, I built them myself — in silence, with effort, late at night, when no one was watching.

So ask yourself: what's harder?
Learning like everyone else and ending up like everyone else?
Or making the decision to learn for yourself — and becoming something else entirely?

Let me put it simply.

> Do you want to believe in success,
> or follow the path that leads to it?

Everything you've done so far — it shouldn't define you. It should free you.

Becoming excellent at deep learning is powerful. But don't stop there.

Go beyond excellence.
Go beyond the hype.

What will make the difference is what you did the moment you opened this notebook:

**You took your situation into your own hands.**

You've learned the *what*. You've built the *how*.
Now it's time to chase the *why*.

So play the game.
Make the effort.
And see how far it can take you.

# Only you can decide that.