

🔧 Step-by-Step: Building an AND Gate

📖 Table of Contents

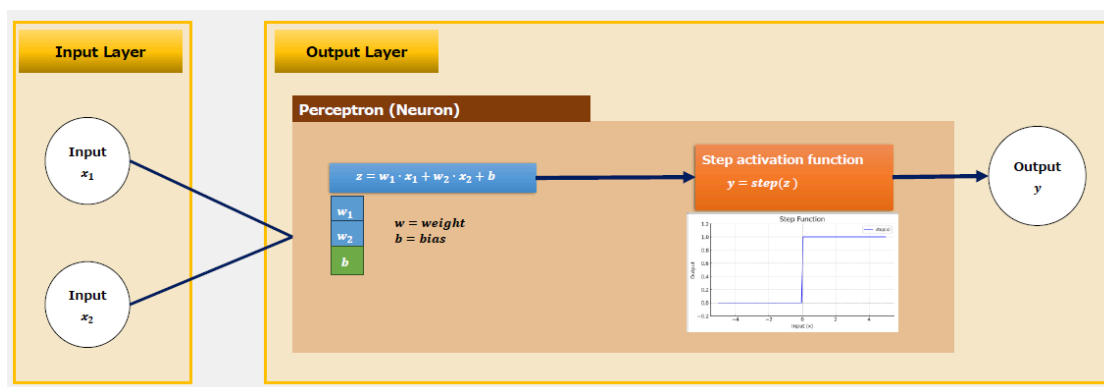
😞 Step 0: What Is a Perceptron?	1
🧠 Step 1: How Does a Perceptron Learn?	2
🔑 Step 2 [Hands-on]: AND Gate Perceptron Learning from Scratch	3
✂ Step 3.1: What Is the Decision Boundary?	4
🔄 3.1.1: Training Process Analogy	4
✅ 3.1.2: When is learning “done”?	4
✂ Step 3.2: How to Derive the Decision Boundary	5
🔑 3.2.1: Compute the decision boundary based on the linear model	5
🖨 Step 4 [Hands-on]: Visualizing the Decision Boundary	6

😞 Step 0: What Is a Perceptron?

A **perceptron** is one of the simplest types of artificial neural networks. It accepts one or more binary inputs and produces a single binary output.

It computes a **weighted sum** of the inputs, adds a **bias**, and then applies an **activation function** (typically a step function) to determine the output.

Diagram:



Mathematically:

$$\text{Output} = \text{step}(w_1 \cdot x_1 + w_2 \cdot x_2 + \dots + b)$$

Where:

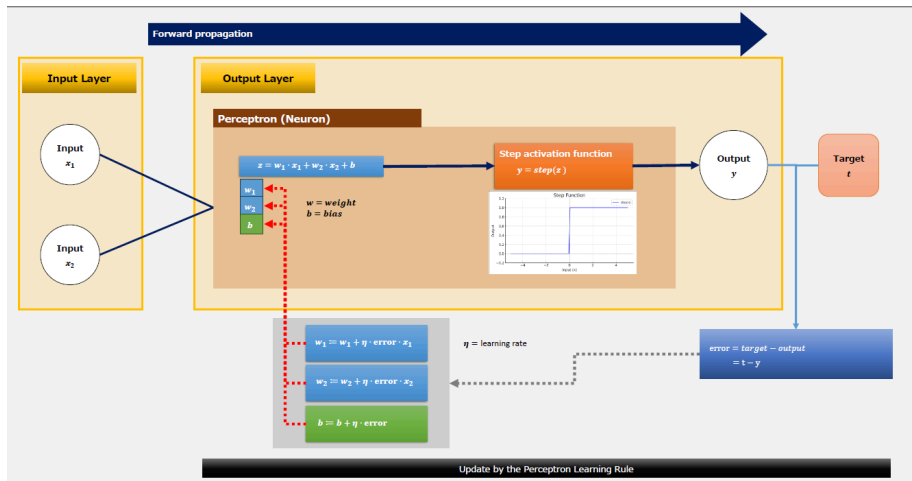
- x_1, x_2 : input values
- w_1, w_2 : corresponding weights
- b : the bias term (a constant added to shift the activation)
- $\text{step}()$: the activation function
 - return 1 if the result is ≥ 0 , otherwise 0

🔍 Why is the Perceptron important?

- It's the **foundation of modern neural networks**
- It can solve problems that are **linearly separable** (like the AND gate)
- It helps us understand **how machines can actually learn**

Step 1: How Does a Perceptron Learn?

This diagram summarizes how a simple perceptron learns from data through **forward propagation**, followed by **weights and bias updates** using the **Perceptron Learning Rule**.



Key Formula:

- $z = w_1 \cdot x_1 + w_2 \cdot x_2 + b$
- $y = \text{step}(z)$
- $\text{error} = t - y$
- $w_1 = w_1 + \eta \cdot \text{error} \cdot x_1$
- $w_2 = w_2 + \eta \cdot \text{error} \cdot x_2$
- $b = b + \eta \cdot \text{error}$

Forward propagation:

- The input values (x_1, x_2) are multiplied by their corresponding weights and added to the bias (b).
- The result is passed through a **step activation function** producing a binary output (y).

Updating weights and bias:

- The model compares this output to the target (t) and computes the error.
- Using the **Perceptron Learning Rule**, the weights and bias are updated to reduce the error.

Note: Backpropagation Is Not Needed Here!

The AND gate can be modeled using a **single-layer perceptron**, which means we don't need to use **backpropagation**.

Instead, we can directly update the weights and bias using the **Perceptron Learning Rule**, which is much simpler and more intuitive.

This is one of the reasons why the AND gate is a great starting point for learning how neural networks work.

Note: Where Does the Perceptron Learning Rule Come From?

The Perceptron Learning Rule wasn't learned by the machine itself — it was designed by researchers to guide how to update weights based on prediction error.

It's a hand-crafted learning algorithm that works well for linearly separable problems, and it served as one of the earliest foundations of machine learning theory.


Key Characteristics:

- For **linearly separable datasets**, the perceptron is **guaranteed to converge**, finding a solution in a **finite number of steps**.
- However, it **cannot solve non-linearly separable problems** (such as the **XOR problem**). This limitation led to the development of **multi-layer perceptrons (MLPs)** to handle more complex patterns.

Step 2 [Hands-on]: AND Gate Perceptron Learning from Scratch

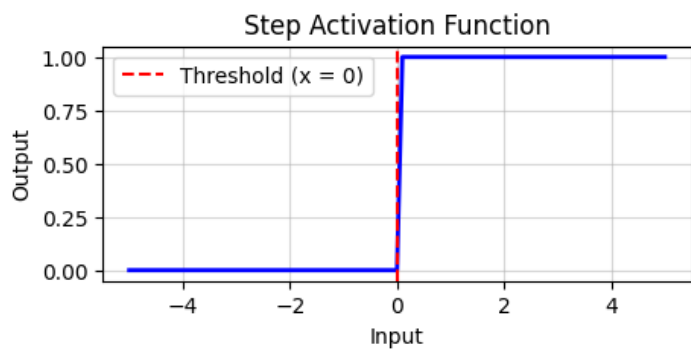
This is a hands-on part of the project.

To try it yourself, please open the notebook below:

 Step02_ANDGatePerceptronLearning.ipynb


By the end of this step, you will be able to:

- [✓] Define a **step function as an activation function**
- [✓] Implement the **Perceptron learning algorithm** from scratch



 Model Test Before Training:

```
test_model()
✓ 0.0s


===  Model Test Results ===
Input: [1, 1], Prediction: 0, Expected: 1 → ✗ Incorrect
Input: [1, 0], Prediction: 0, Expected: 0 → ✓ Correct
Input: [0, 0], Prediction: 0, Expected: 0 → ✓ Correct
Input: [0, 1], Prediction: 0, Expected: 0 → ✓ Correct

Accuracy: 75.0% (3/4)

Final Weights: w1 = 0.279, w2 = -0.950
Final Bias: -0.450
```

 Model Test After Training:

```
test_model()
✓ 0.0s

===  Model Test Results ===
Input: [1, 1], Prediction: 1, Expected: 1 → ✓ Correct
Input: [0, 1], Prediction: 0, Expected: 0 → ✓ Correct
Input: [1, 0], Prediction: 0, Expected: 0 → ✓ Correct
Input: [0, 0], Prediction: 0, Expected: 0 → ✓ Correct

Accuracy: 100.0% (4/4)

Final Weights: w1 = 0.279, w2 = 0.250
Final Bias: -0.450
```

✂ Step 3.1: What Is the Decision Boundary?

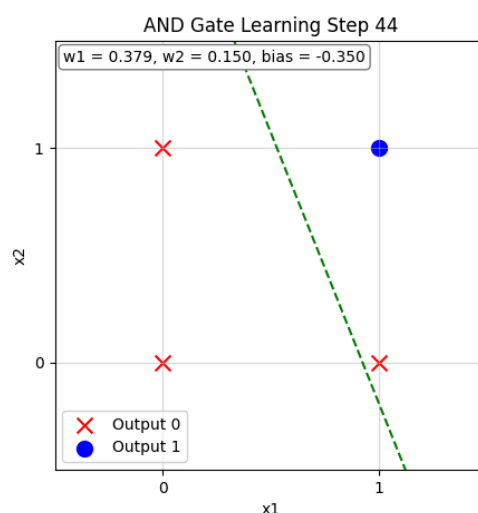
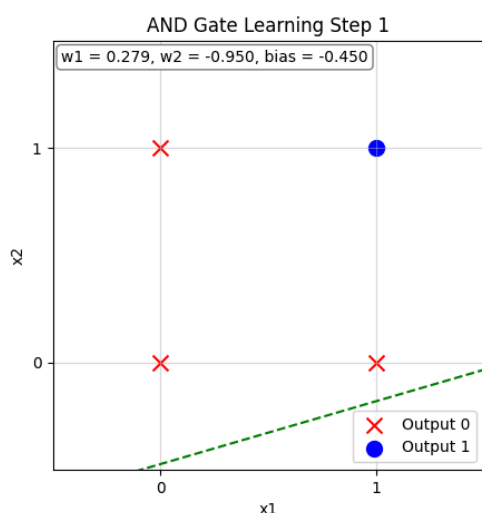
A **decision boundary** is a line (or a surface) that a machine learning model draws to **separate different classes** in a classification problem.

🔄 3.1.1: Training Process Analogy

When the model starts learning, it doesn't know where to draw the line — it's kind of guessing.

But during training, the model keeps adjusting the line based on the data it sees.

It tries to **reduce the number of mistakes**, and slowly **moves the boundary to better separate the classes**.



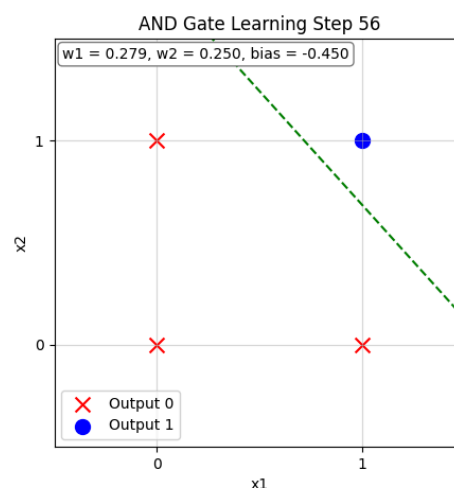
✅ 3.1.2: When is learning “done”?

Learning is complete where:

- The decision boundary stops changing much
- The loss becomes stable (converges)
- The model performs well (like accuracy is high)

In other words:

The model has learned a good way to separate the classes.



📖 Note: Types of Decision Boundaries During Training:

A decision boundary exists at every stage of training, even if the model hasn't fully learned yet. It reflects the current parameters and evolves over time.

Training Stage	Boundary Type	Description
Before or Early Training	Initial Decision Boundary	Parameters are untrained or randomly initialized
During Training	Intermediate Decision Boundary	Continuously updated , but still inaccurate
After Training Completion	Final Decision Boundary	Parameters have converged ; classification is stable

✂ Step 3.2: How to Derive the Decision Boundary

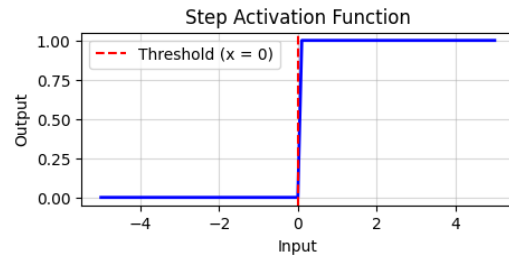
✎ 3.2.1: Compute the decision boundary based on the linear model

The model computes a weighted sum (linear combination) of inputs:

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

The predicted class is determined by the sign of z :

- if $z \geq 0 \rightarrow$ class 1
- if $z < 0 \rightarrow$ class 0



To draw the decision boundary, solve for $z = 0$:

$$w_1 \cdot x_1 + w_2 \cdot x_2 + b = 0$$

When $w_2 \neq 0$ (General Case)

$$w_1 \cdot x_1 + w_2 \cdot x_2 + b = 0$$

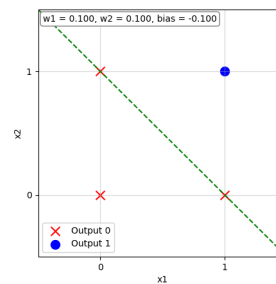
$$w_2 \cdot x_2 = -(w_1 \cdot x_1 + b)$$

$$x_2 = \frac{-(w_1 \cdot x_1 + b)}{w_2}$$

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{b}{w_2}$$

→ This gives a straight line with slope $-\frac{w_1}{w_2}$

Simple Example:



- $w_1 = 0.1, w_2 = 0.1, b = -0.1$

$$x_2 = -\frac{0.1}{0.1}x_1 - \frac{(-0.1)}{0.1} = -x_1 + 1$$

$$\rightarrow x_2 = -x_1 + 1$$

When $w_2 = 0$ and $w_1 \neq 0$ (Special Case)

$$w_1 \cdot x_1 + w_2 \cdot x_2 + b = 0$$

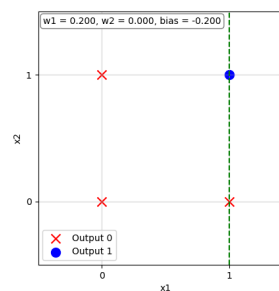
$$w_1 \cdot x_1 = -b$$

$$x_1 = \frac{-b}{w_1}$$

$$x_1 = -\frac{b}{w_1}$$

→ This defines a vertical boundary line

Simple Example:




- $w_1 = 0.2, w_2 = 0.0, b = -0.2$

$$x_1 = \frac{-(-0.2)}{0.2} = 1$$

$$\rightarrow x_1 = 1$$

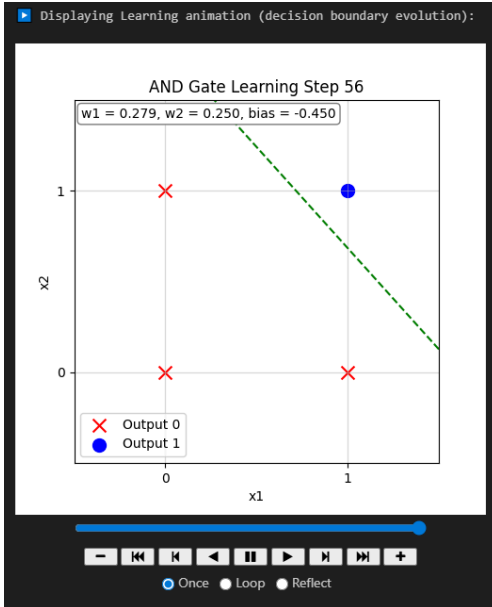
Step 4 [Hands-on]: Visualizing the Decision Boundary

This is a hands-on part of the project.
To try it yourself, please open the notebook below:

 [Step04_DecisionBoundary.ipynb](#)

By the end of this step, you will be able to:

- [✓] Plot the decision boundary based on the learned weights and bias



That concludes my explanation.