# Introduction to Neural Networks

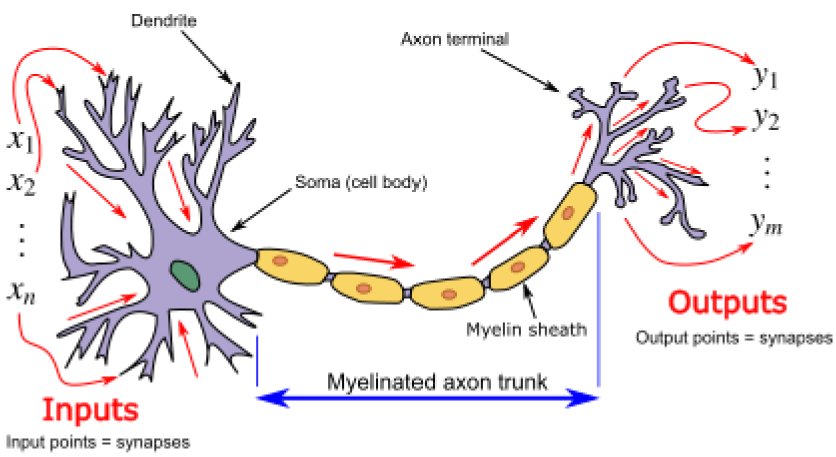
**Deep learning architectures**—such as **Recurrent Neural Networks (RNNs)**, **Convolutional Neural Networks (CNNs)**, and **Deep Belief Networks (DBNs)**—have been widely applied across various domains, including *computer vision*, *speech recognition*, *natural language processing*, *machine translation*, *bioinformatics*, *drug discovery*, and *medical image analysis*.

What sets deep learning apart is the **depth of computation**, which enables these models to learn and represent **complex and hierarchical patterns**—a key requirement for tackling the most challenging real-world datasets.

**Deep learning models** are built upon the foundation of **Artificial Neural Networks (ANNs)**, which are inspired by the **structure and functioning of the human brain**.

In biology, the brain excels at solving complex problems, thanks to its network of specialized cells called **neurons** (or **nerve cells**). These neurons are the **fundamental building blocks** of the brain.

Some neurons are responsible for **receiving sensory input** from the external environment, while others **transmit motor commands** to control muscle movements.




Each neuron consists of four main parts: the **soma** (cell body), **dendrites**, **axon**, and **synapses**.

- **Dendrites:** These branch-like structures receive signals from multiple neighboring neurons.

- **Soma:** Also known as the cell body, it collects and integrates the incoming signals from the dendrites.
- **Axon:** Responsible for transmitting electrical signals away from the neuron's cell body.
- **Axon Terminals (Synapses):** These are the endpoints of the axon. They form connections with other neurons and are responsible for transferring information to those neurons through chemical or electrical signals.
- If the combined signal received by the soma is strong enough, it triggers an electrical impulse.
- This impulse is then transmitted through the axon to other neurons via synapses.

Although **biological brains** are capable of solving highly complex problems, each **neuron** contributes by handling only a **small part** of the overall task.

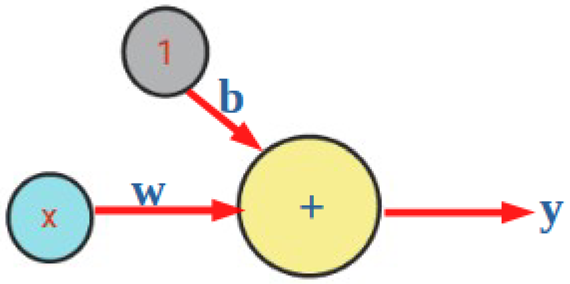
Similarly, in **artificial neural networks**, the system is composed of many units called **artificial neurons**, which work together to achieve a desired output.

Each **artificial neuron** performs only a **simple computation**, but when combined in large numbers, they can model and solve very complex problems.

# Artificial Neuron (Linear Unit)

An **Artificial Neuron** (**Linear Unit**) takes an input, denoted by **x**.

- Each input is connected to the neuron with an associated **weight**, represented by **w**.
- When the input flows through this connection, it is **multiplied by the weight**.
- So, the value that reaches the neuron is $\mathbf{w} \times \mathbf{x}$.
- A **neural network learns** by adjusting these weights based on the error in its predictions.

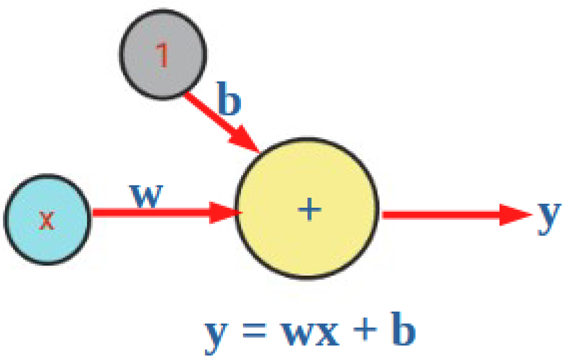



The **bias** (denoted as **b**) is a special type of **weight** in a neural network.

- Unlike other weights, the **bias** is not associated with any input data.
- To represent it in diagrams, we typically use a constant input of **1**, so the value reaching the neuron from the bias is simply **b** (since $1 \times b = b$).
- The bias allows the neuron to **adjust the output independently** of the inputs, enhancing the flexibility of the network.

The value produced by the neuron is called the **output**, denoted as **y**.

- The neuron computes this by **summing all the signals** it receives through its connections.
- For a single input, the output is calculated as: $\mathbf{y = w \times x + b}$.
- The **bias term (b)** allows the function to **shift left or right**, enhancing flexibility.
- Thus, an **artificial neuron** acts as a **mathematical function** that transforms input into output.




In artificial neural networks, the **connections between neurons** are represented by **weights**, simulating the behavior of biological neurons.

- **Positive weights** represent **excitatory connections**, which promote activation.

- **Negative weights** represent **inhibitory connections**, which suppress activation.
- Each input is **multiplied by its weight**, and the results are **summed together**.
- This process is referred to as a **linear combination**.

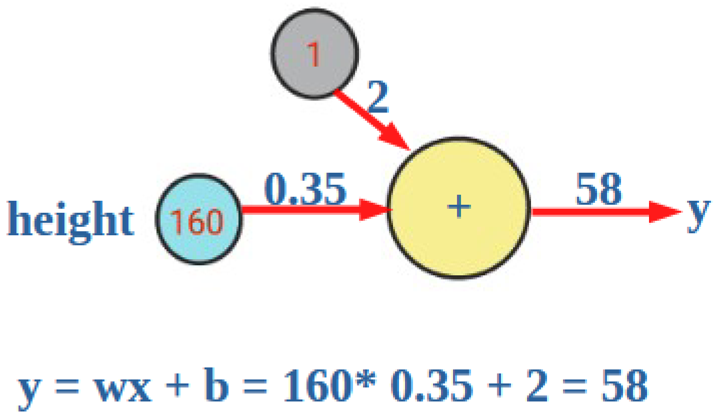
# Linear Unit as a Model

The **Linear Unit** can be used to model simple relationships. For example, let's consider a case where:

- **Input:** Height (in cm)
- **Output:** Weight (in kg)

After training the model, suppose we get the following parameters:

- **Weight (w):** 0.35
- **Bias (b):** 2




To estimate the weight of a person who is **160 cm** tall:

**Output = w × input + b = 0.35 × 160 + 2 = 58 kg**

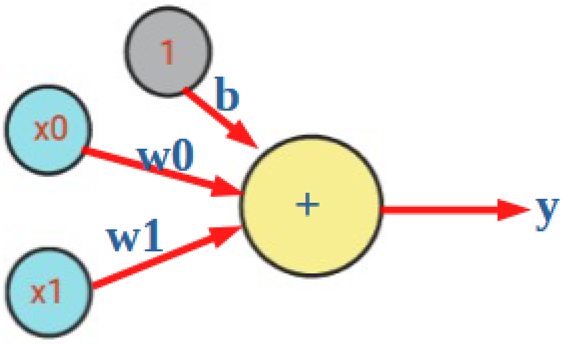
So, the predicted weight is **58 kg**.

# Multiple Inputs in a Linear Unit

Suppose the dataset includes an additional feature, such as **gender**. We can simply add another input connection to the neuron.

- Each input is multiplied by its corresponding **weight**.
- All the weighted inputs are then **added together** to produce the output.

This approach allows the model to use **multiple features** for prediction.

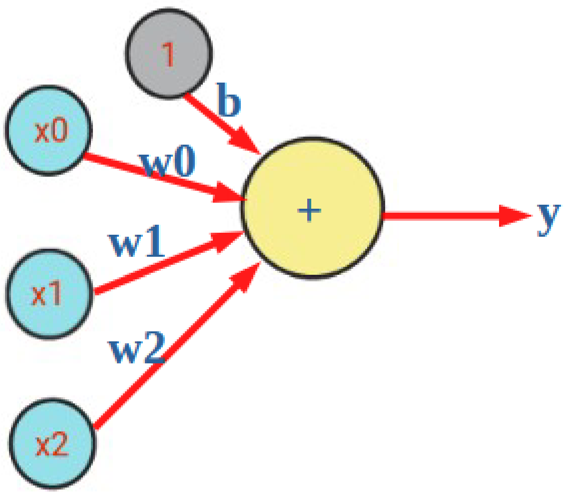



$$y = w0x0 + w1x1 + b$$

Suppose the dataset includes three input features: **height**, **gender**, and **country**. We can add one input connection to the neuron for each feature.

- Each feature is multiplied by its corresponding **weight**.
- All the weighted inputs are **summed** to compute the final output.

This allows the neuron to process **multiple (three) input features** in the prediction task.




$$y = w0x0 + w1x1 + w2x2 + b$$

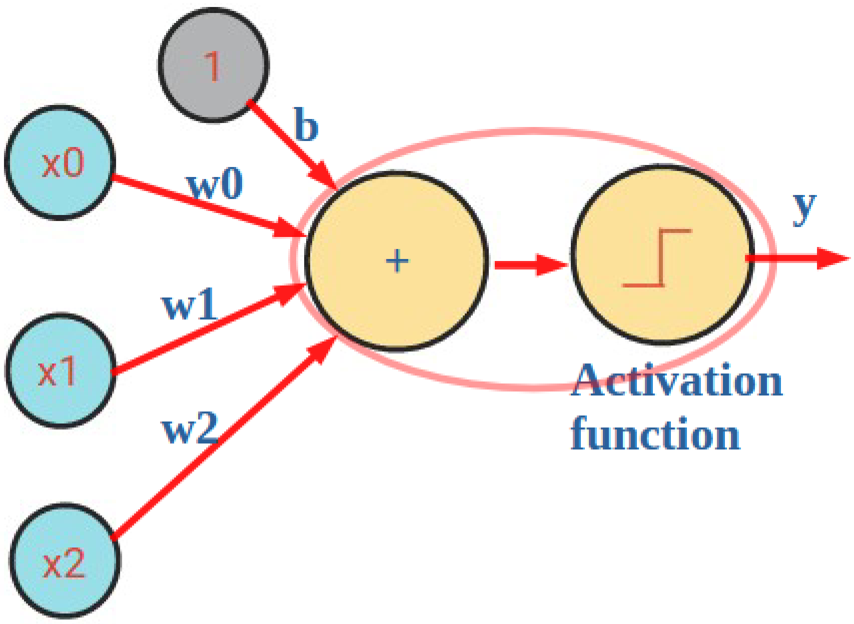
# Perceptron

The **Perceptron** was developed in **1958** by **Frank Rosenblatt**, based on earlier research by **Warren McCulloch** and **Walter Pitts**.

- A perceptron is an **artificial neuron** that uses the **Heaviside step function** as its activation function.
- It is used in **supervised learning** to train **binary classifiers**.
- A binary classifier is a function that determines whether a given input vector **belongs to a specific class**.

# Activation Function

The **activation function** is applied to the **weighted sum of the inputs** to produce the neuron's output.



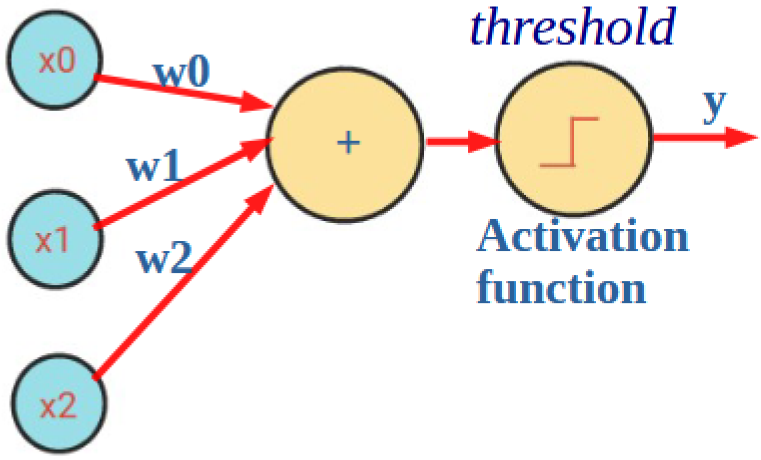

- It maps the weighted sum to an output value, usually in the range of **0 to 1**.
- It determines whether the neuron should be **activated** or not, hence the name.

# How a Perceptron Works

In this example, the **perceptron** has three inputs: $x_0$, $x_1$, and $x_2$. In general, a perceptron can accept any number of inputs.

To determine the output, **Frank Rosenblatt** introduced a simple rule:

- Each input $x_i$ is assigned a weight $w_i$, representing the importance of that input.
- The perceptron computes the **weighted sum**: $\sum_i w_i \times x_i$
- This sum is compared to a predefined **threshold** (a real number).




The output is then calculated as:

$$f(x)=\begin{cases} 0 & \text{if} \sum_i w_i . x_i \leq threshold \\ 1 & \text{if} \sum_i w_i . x_i > threshold \end{cases}$$

In a perceptron, the **bias** (b) can be defined as the **negative of the threshold**:

```
b ≡ -threshold
```

This reformulation simplifies the perceptron's computation, allowing us to express the decision rule without explicitly using the threshold.

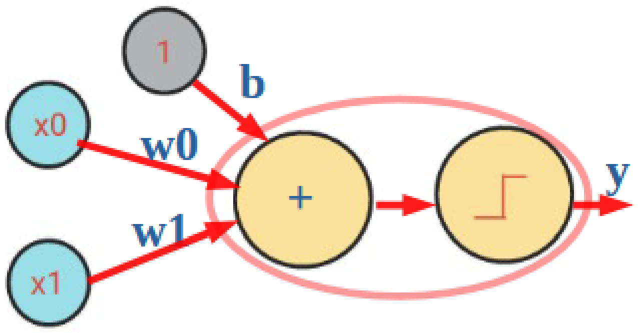
$$f(x)=\begin{cases} 0 & \text{if } w . x + b \leq 0 \\ 1 & \text{if } w . x + b > 0 \end{cases}$$

That's all there is to the basic working of a perceptron!

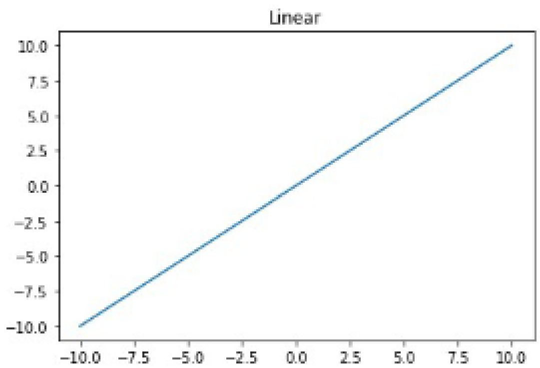
When a perceptron has **two inputs**, its input space can be visualized as a **2D plane**.

- Each input represents a point on this 2D plane.

- The neuron forms a **decision boundary**, which appears as a **straight line** (also called a *subspace* or *hyperplane* in higher dimensions) dividing the space into different classes.

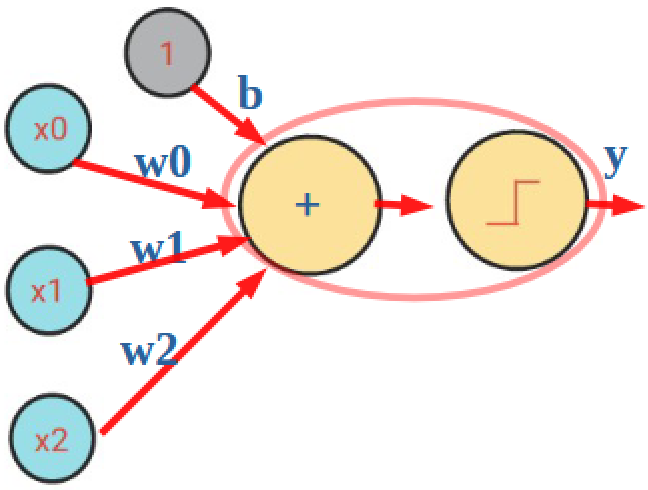



**w0x0 + w1x1 + b = 0**

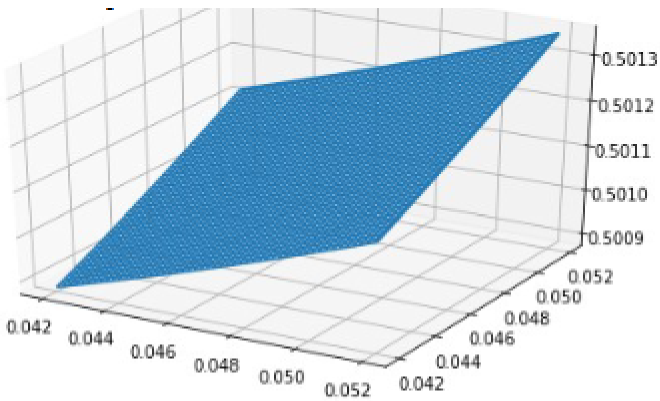



When a perceptron has **three inputs**, the input space becomes a **3D space**.

- Each input corresponds to a point in this 3D space.
- The neuron creates a **decision boundary** that can be visualized as a **2D plane**—also referred to as a *hyperplane* or *subspace*—which separates different regions in the 3D input space.



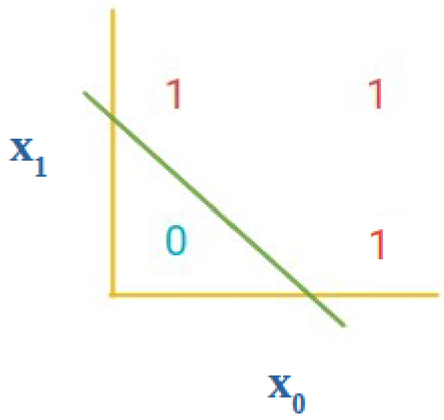

$$w0x0 + w1x1 + w2x2 + b = 0$$




# Solving Logical OR Using a Linear Perceptron

## OR Truth Table

| Input $x_0$ | Input $x_1$ | OR Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |




## Perceptron Function

The perceptron computes output using:
`y = 1 if (w₀x₀ + w₁x₁ + b > 0), else 0`

## Choose Weights and Bias

- $w_0 = 0.5$
- $w_1 = 0.5$
- $b = -0.25$

The function becomes: `y = 1 if (0.5 * x₀ + 0.5 * x₁ - 0.25 > 0)`

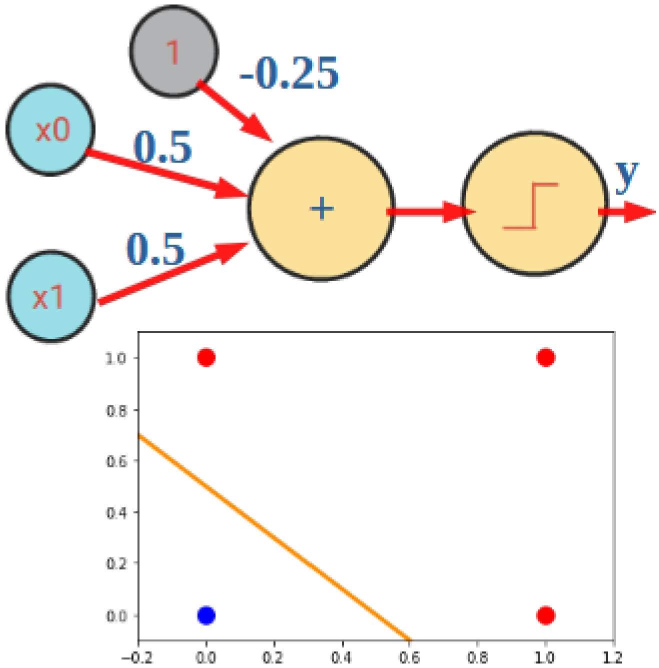
## Test with Inputs

| $x_0$ | $x_1$ | $0.5x_0 + 0.5x_1 - 0.25$ | Output |
|---|---|---|---|
| 0 | 0 | −0.25 | 0 |
| 0 | 1 | 0.25 | 1 |
| 1 | 0 | 0.25 | 1 |
| 1 | 1 | 0.75 | 1 |

## Geometric View

In the 2D input space, the perceptron forms a linear decision boundary defined by:

`0.5x₀ + 0.5x₁ - 0.25 = 0`
Simplified as: `x₀ + x₁ = 0.5`




This line separates the input `(0, 0)`, which produces output 0, from the others like `(0,1)`, `(1,0)`, and `(1,1)` which lie above the line and produce output 1.

### Why It Works?

- Logical OR is a **linearly separable function**.
- A single-layer perceptron is sufficient to classify OR correctly.

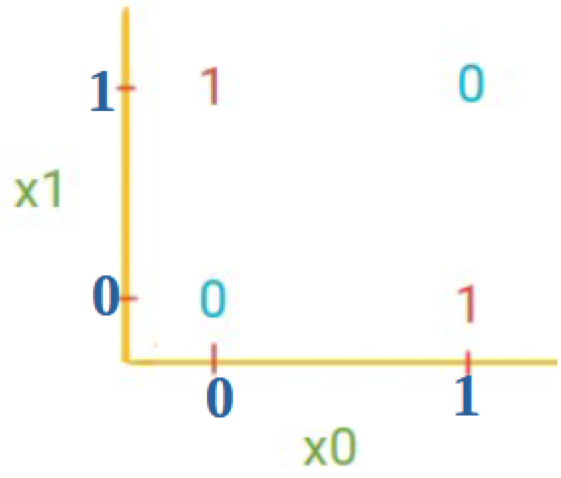
# Multi Layer Perceptron

Single-layer perceptron can solve linearly separable problems like `Logical OR`, but not nonlinear problems like `XOR (Exclusive OR)`.

### XOR Truth Table

| Input $x_0$ | Input $x_1$ | XOR Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |

| | | |
|---|---|---|
| 1 | 1 | 0 |



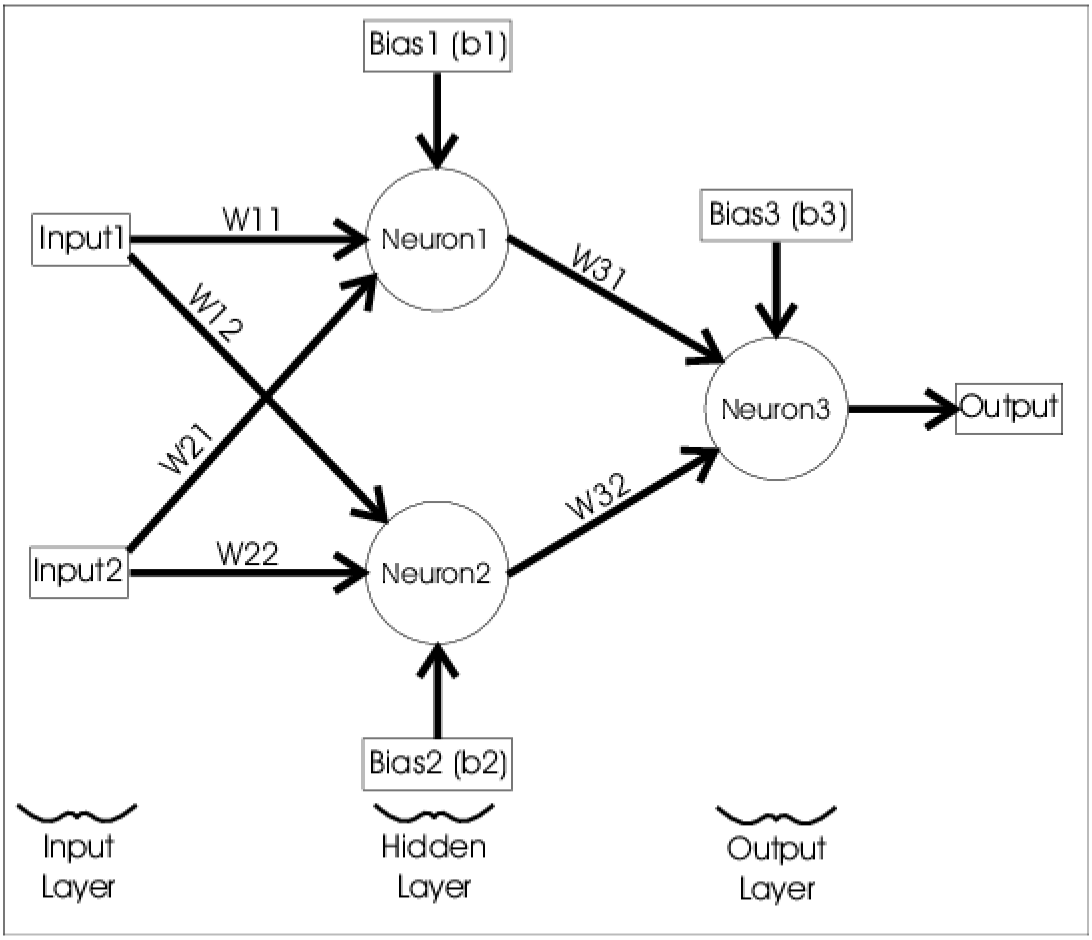

There's no straight line that can separate the 1s from the 0s — this makes XOR non-linearly separable. Therefore, a linear decision boundary created by a single-layer perceptron won't work.

We can combine multiple perceptrons in layers to learn non-linear decision boundaries. This setup is called a **Multi-Layer Perceptron (MLP)**.

## Network Architecture

- **2 input neurons** ($x_0$, $x_1$)
- **1 hidden layer** with 2 neurons
- **1 output neuron**

This architecture allows the network to model non-linear functions like XOR, which are not linearly separable.




The hidden layer learns intermediate representations that transform the input space into a form where a linear classifier (like the output neuron) can separate the classes correctly.

## Inputs

Let's denote the input vector as:

$x = [x_0 \; x_1]$

## First Layer (Hidden Layer)

- **Weights:**

```
W = [1 1
     1 1]
```

- **Biases:**

```
c = [0
    -1]
```

## Second Layer (Output Layer)

- **Weights:**

  ```
  w = [1 -2]
  ```

- **Bias:**

  ```
  b = 0
  ```

In hidden layers let us use step activation function

### Step Activation Function

```
step(z) = { 1 if z > 0, 0 otherwise }
```

## 1. Input: (0, 0)

### Hidden Layer Output:

```
z = W · x + c     =
   [1  1]      [0]     [ 0]
   [1  1]  .   [0]  +  [-1]
   = [ 0]
     [-1]

step(z) =     [0]
              [0]
```

### Output Layer:

```
y = w  · h + b  =   [ 1  -2 ]   ·   [0]   = 0
                                    [0]

Output = 0
```

## 2. Input: (0, 1)

### Hidden Layer Output:

```
z = W · x + c     =
   [1  1]      [0]      [ 0]
   [1  1]   .  [1]  +  [-1]
   = [1]
     [0]

step(z) =     [1]
              [0]
```

### Output Layer:

```
y = w  · h + b  =   [ 1  -2 ]   ·   [1]   = 1
                                    [0]

Output = 1
```

## 3. Input: (1, 0)

### Hidden Layer Output:

```
z = W · x + c     =
   [1  1]      [1]      [ 0]
   [1  1]   .  [0]  +  [-1]
   = [1]
     [0]

step(z)  =    [1]
              [0]
```

### Output Layer:

```
y = w  · h + b  =   [ 1  -2 ]   ·   [1]   = 1
                                    [0]
```

```
Output = 1
```

## 4. Input: (1, 1)

### Hidden Layer Output:

```
z = W · x + c     =
   [1  1]      [1]     [ 0]
   [1  1]  .   [1]  +  [-1]
   = [2]
     [1]

step(z)  =    [1]
              [1]
```

### Output Layer:

```
y = w  · h + b  =   [ 1  -2 ]   ·   [1]   = -1
                                    [1]

Output = 0
```

## Final Output (XOR Logic):

| $x_0$ | $x_1$ | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |