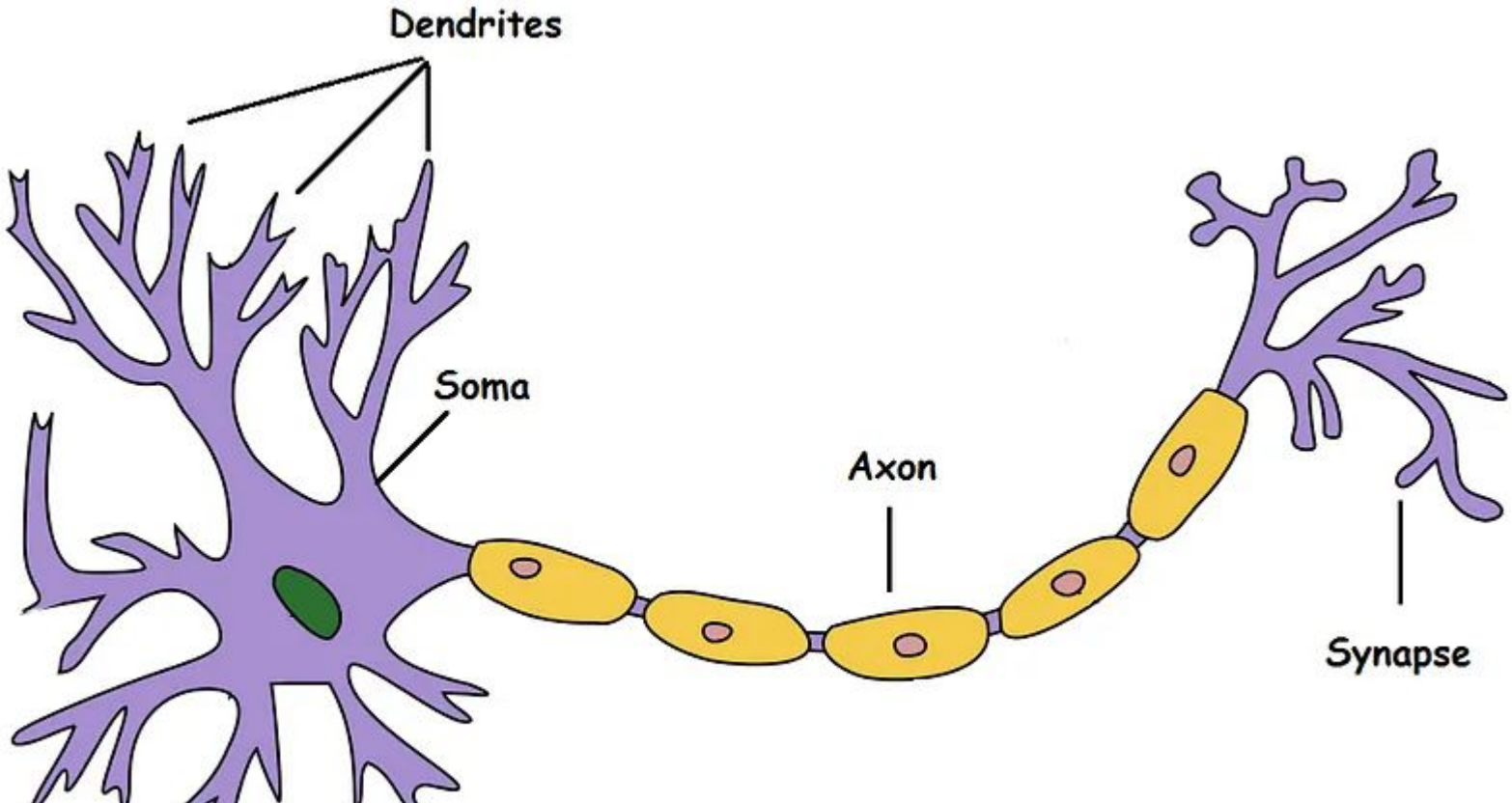
Biological Neuron



Biological Neuron

Dendrite : Receives signals from other neurons

Soma : Processes the information

Axon : Transmits the output of this neuron

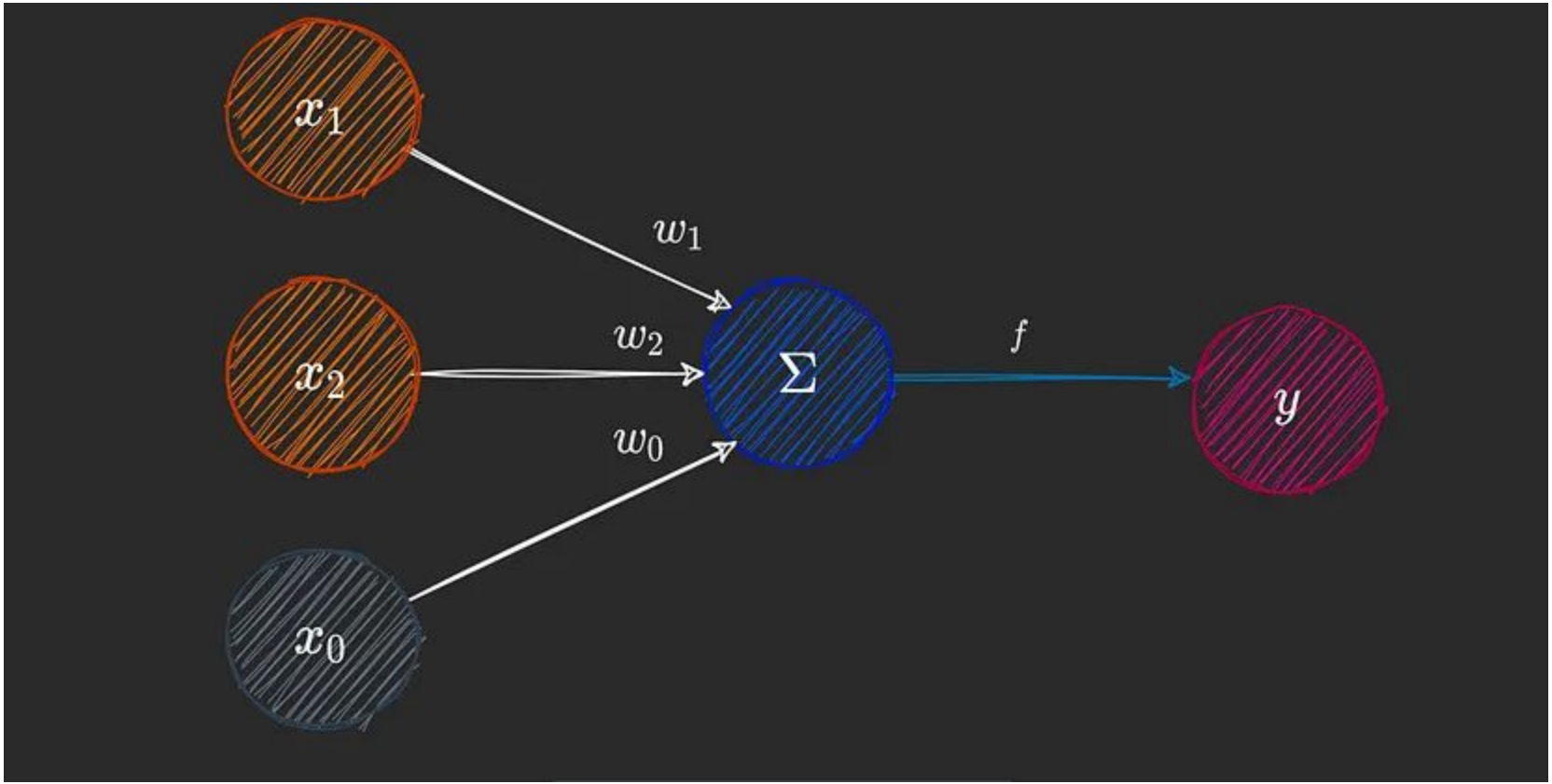
Synapse : Point of connection to other neurons

The perceptron is a classification algorithm. Specifically, it works as a linear binary classifier. It was invented in the late 1950s by Frank Rosenblatt.

The perceptron basically works as a threshold function — non-negative outputs are put into one class while negative ones are put into the other class.

A perceptron has the following components:

- Input nodes
- Output node
- An activation function
- Weights and biases
- Error function


x_1
w_1
x_2
w_2
Σ
f
y
w_0
x_0

Weights and Biases

These parameters are what we update when we talk about “training” a model. They are initialized to some random value or set to 0 and updated as the training progresses.

Evaluation

- Compute the dot product of the input and weight vector
- Add the bias
- Apply the activation function.

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

$$y = f(w \cdot X + b)$$

Activation Function

This function allows us to fit the output in a way that makes more sense. For example, in the case of a simple classifier, an output of say `-2.5` or `8` doesn't make much sense with regards to classification. If we use something called a sigmoidal activation function, we can fit that within a range of 0 to 1, which can be interpreted directly as a probability of a datapoint belonging to a particular class.

Classification

$$Class_1 : w \cdot X + b \geq 0$$

$$Class_2 : w \cdot X + b < 0$$

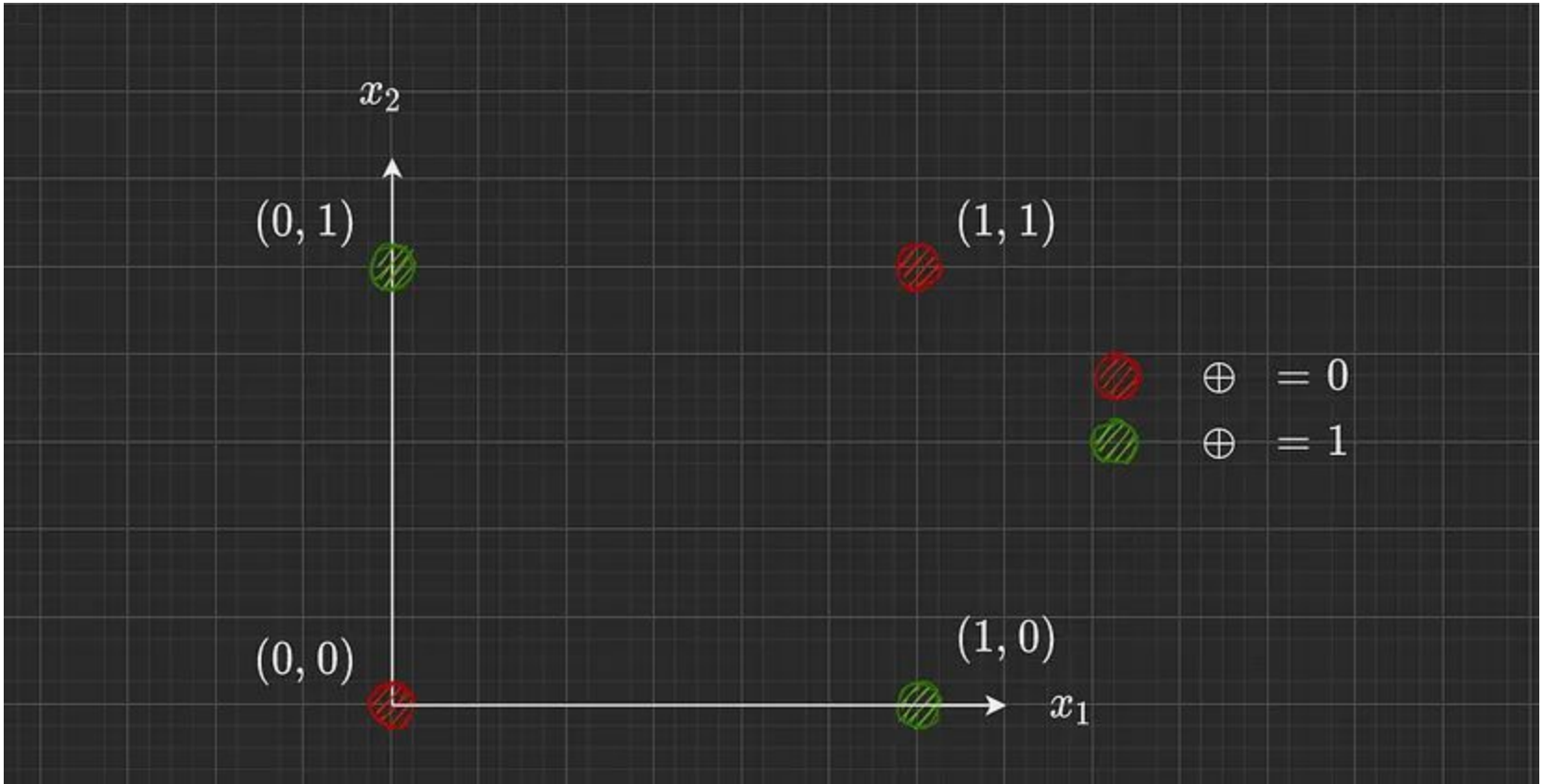
Training

$$\Delta w = node \cdot (actual - computed)$$

$$w_{updated} = w_{old} + lr \cdot \Delta w$$

XOR-truth table

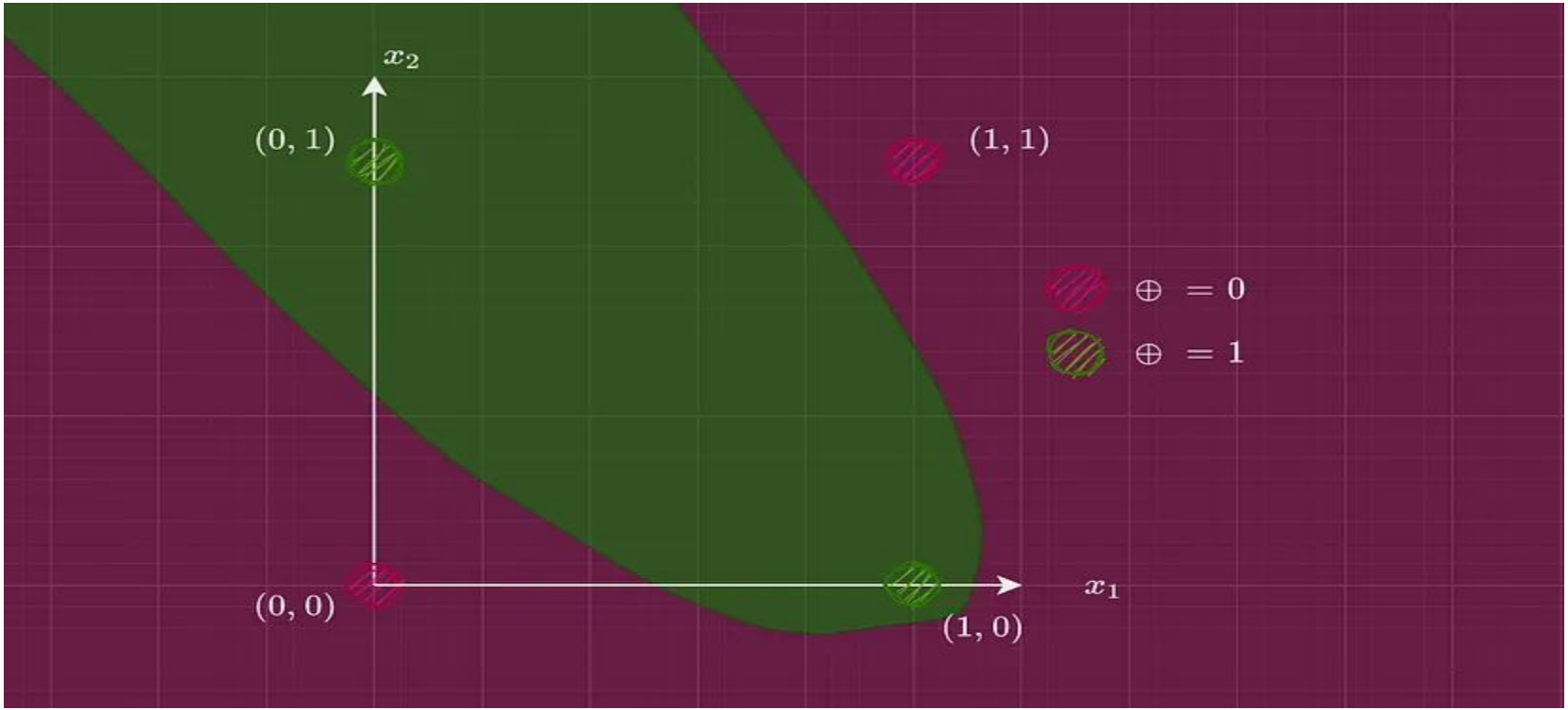
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0


x_2
$(0, 1)$
$(1, 1)$
$\oplus = 0$
$\oplus = 1$
$(0, 0)$
$(1, 0)$
x_1

A perceptron can only converge on linearly separable data.

Therefore, it isn’t capable of imitating the XOR function.

The Need for Non-Linearity



XOR

$$XOR(A,B) = A \cdot \overline{B} + B \cdot \overline{A}$$

$$XOR(A,B) = A \cdot \overline{B} + B \cdot \overline{A} + (A \cdot \overline{A} + B \cdot \overline{B})$$

$$XOR(A,B) = (A \cdot \overline{A} + A \cdot \overline{B}) + (B \cdot \overline{A} + B \cdot \overline{B})$$

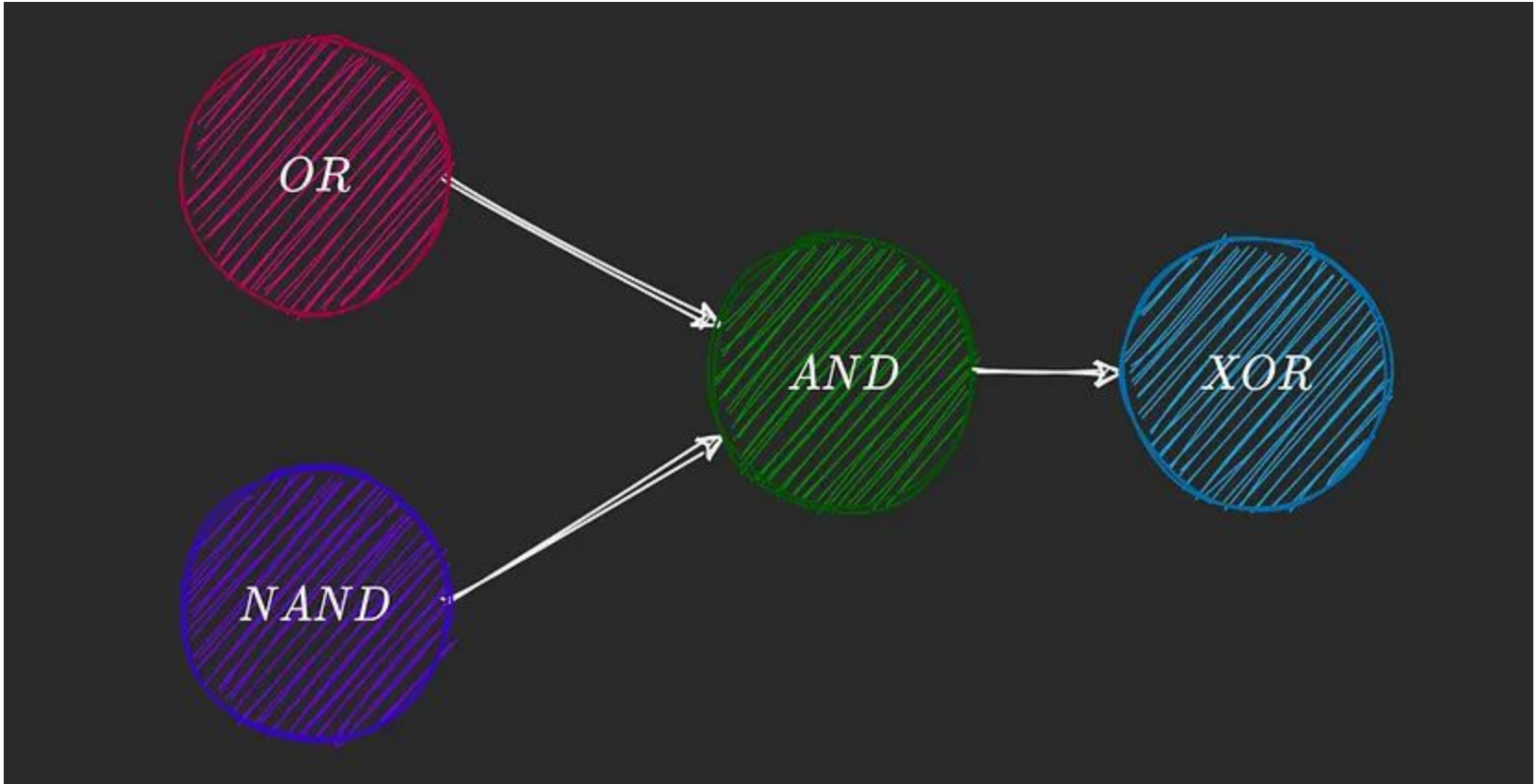
$$XOR(A,B) = (A + B) \cdot (\overline{AB})$$

XOR

The XOR function can be condensed into two parts: **a NAND and an OR**. If we can calculate these separately, we can just combine the results, using **an AND gate.**

$$XOR(x_1, x_2) = (x_1 + x_2) \cdot (\overline{x_1 x_2})$$

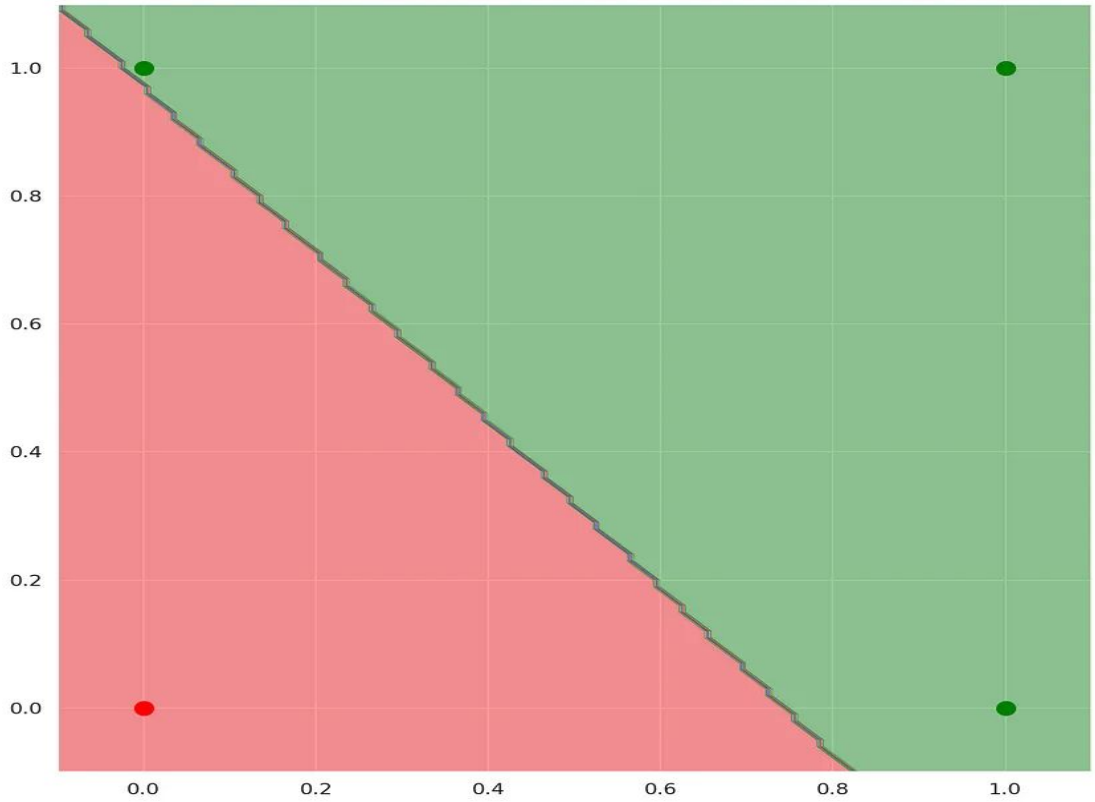
The plan

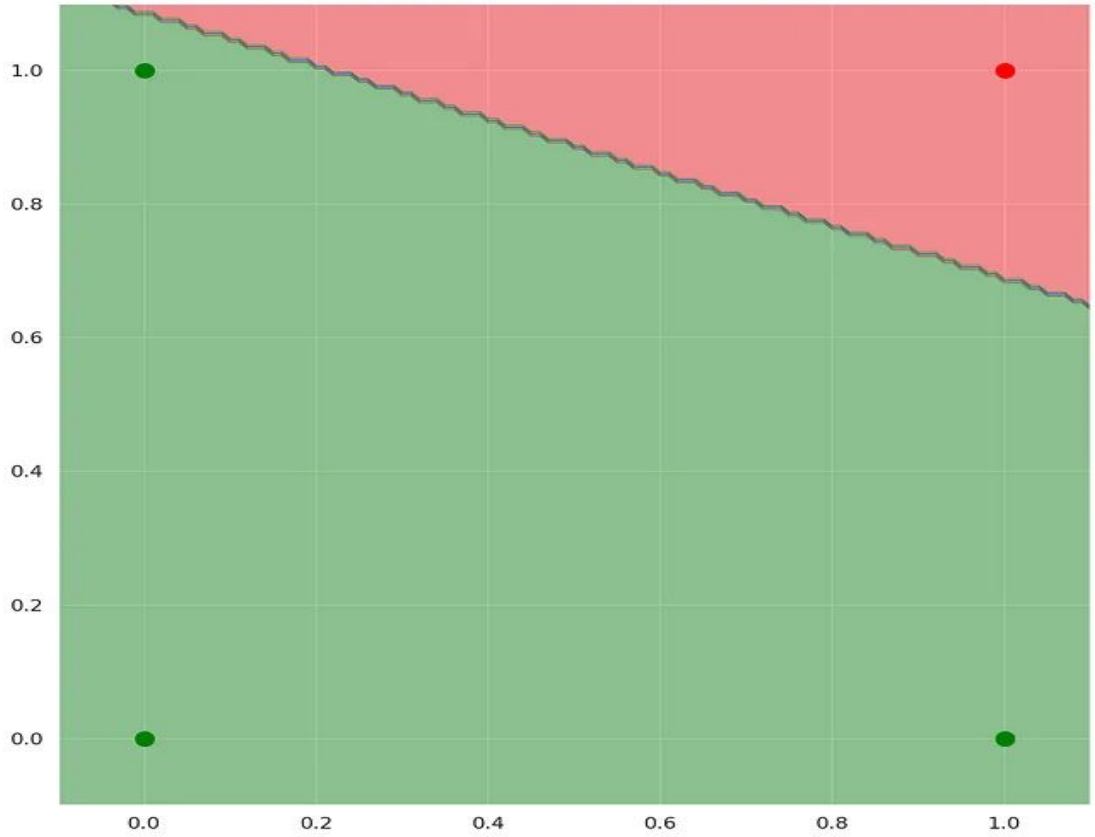


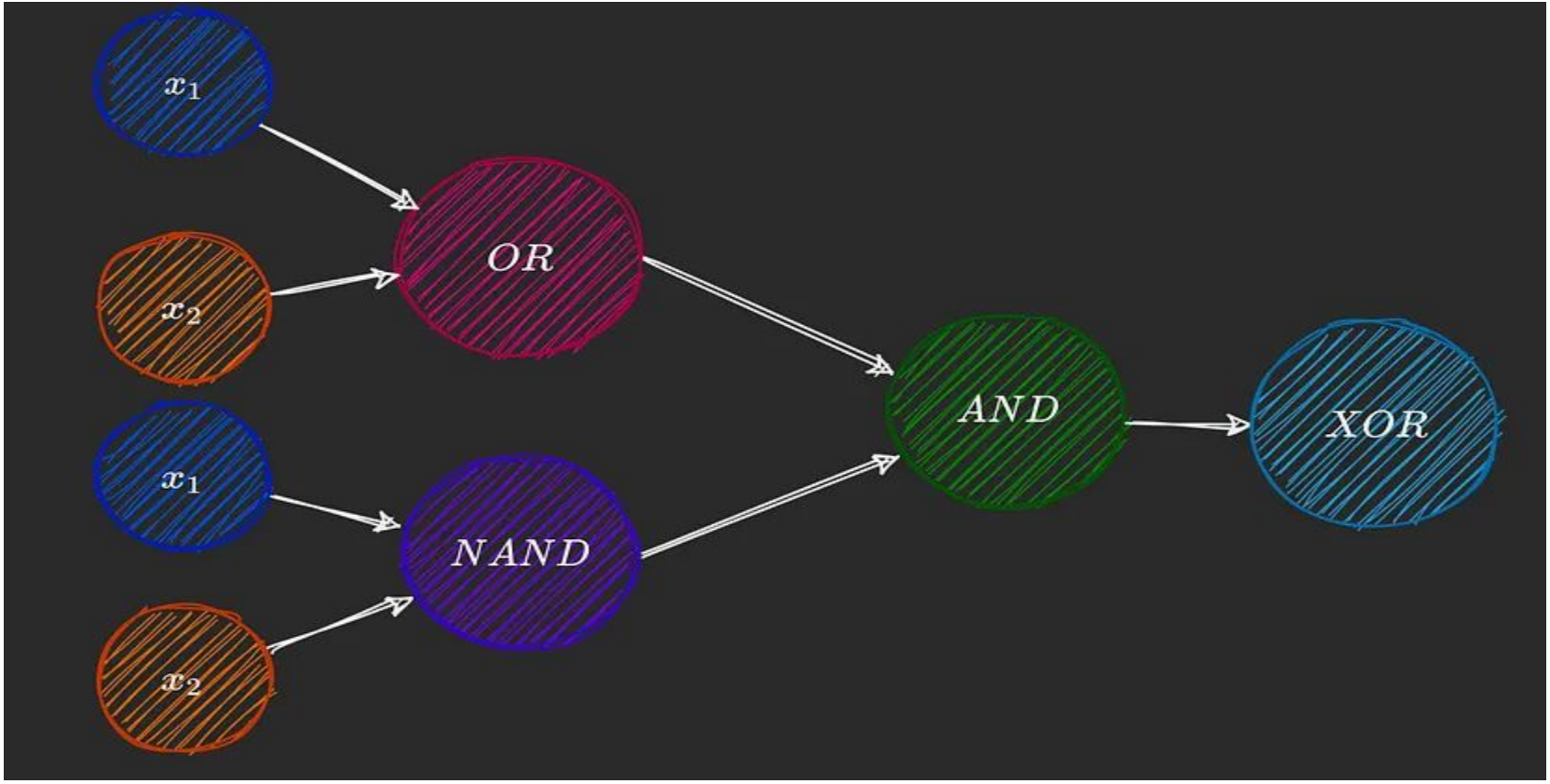
OR Gate

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

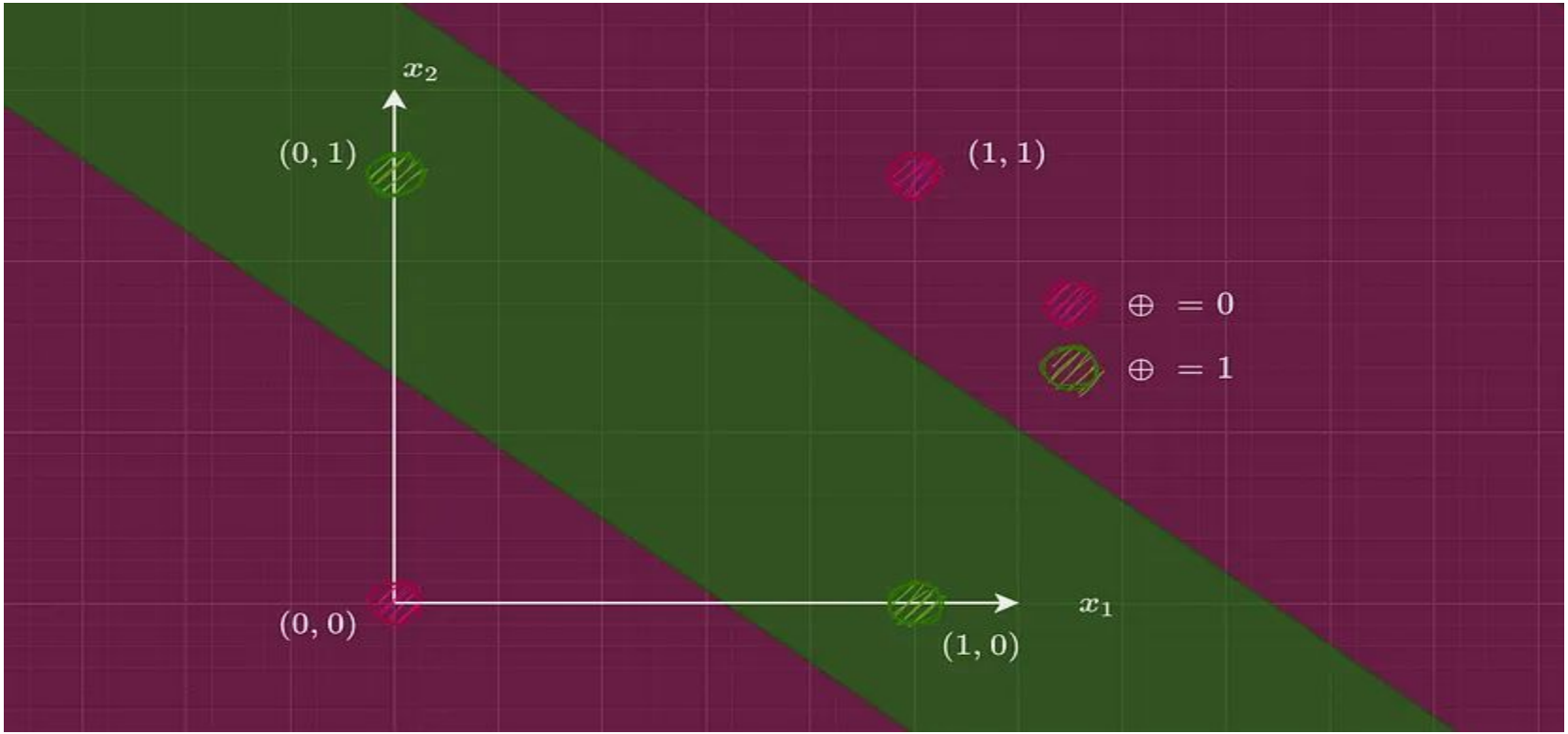
The Output plot of our OR perceptron




1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


x_1
x_2
x_1
x_2
OR
NAND
AND
XOR

The Output plot showing a correct classification on our XOR data



The Multi-layered Perceptron

The overall components of an MLP like input and output nodes, activation function and weights and biases are the same as those we just discussed in a perceptron.

The biggest difference? An MLP can have hidden layers.

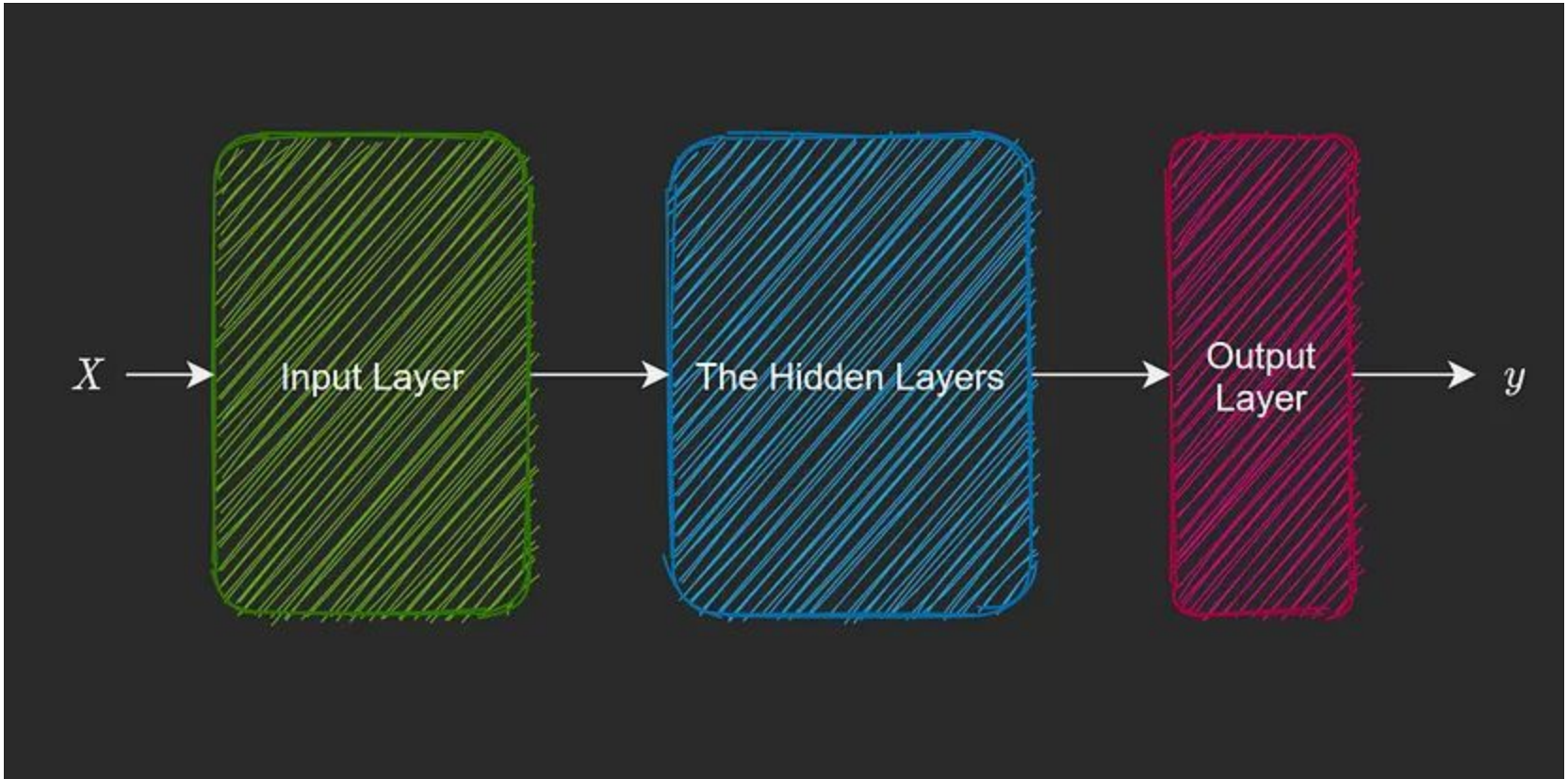
Hidden layers

Hidden layers are those layers with nodes other than the input and output nodes.

> An MLP is generally restricted to having a single hidden layer.

The hidden layer allows for non-linearity. A node in the hidden layer isn't too different to an output node: nodes in the previous layers connect to it with their own weights and biases, and an output is computed, generally with an activation function.

The general structure of a multi-layered perceptron



Backpropogation

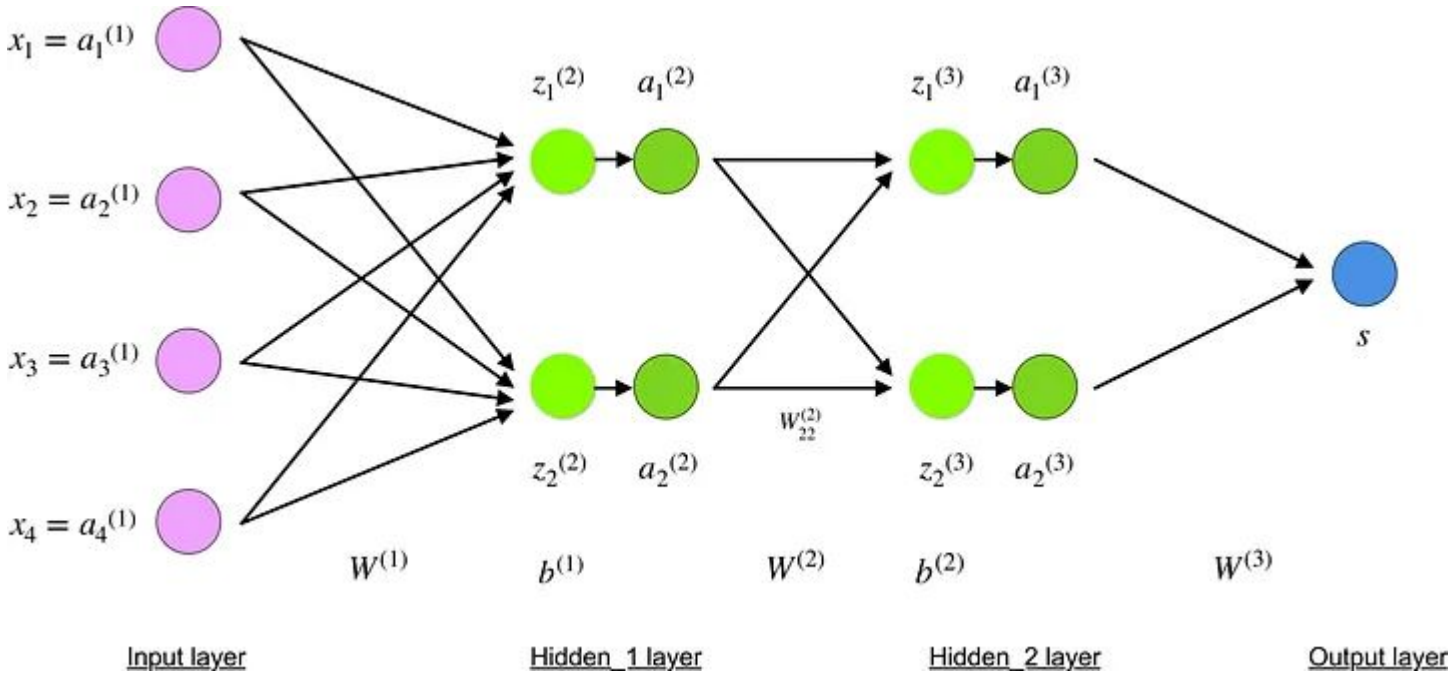
Backpropagation is an algorithm for update the weights and biases of a model based on their gradients with respect to the error function, starting from the output layer all the way to the first layer.

Content taken from

https://medium.com/towards-data-science/understanding-backpropagation-algorithm-7bb3aa2f95fd

Backpropagation algorithm is probably the most fundamental building block in a neural network. It was first introduced in 1960s and almost 30 years later (1989) popularized by Rumelhart, Hinton and Williams in a paper called *“Learning representations by back-propagating errors”*.

The algorithm is used to effectively train a neural network through a method called chain rule. In simple terms, after each forward pass through a network, backpropagation performs a backward pass while adjusting the model’s parameters (weights and biases).


$x_1 = a_1^{(1)}$
$x_2 = a_2^{(1)}$
$x_3 = a_3^{(1)}$
$x_4 = a_4^{(1)}$
$z_1^{(2)}$
$a_1^{(2)}$
$z_2^{(2)}$
$a_2^{(2)}$
$z_1^{(3)}$
$a_1^{(3)}$
$z_2^{(3)}$
$a_2^{(3)}$
s
$W_{22}^{(2)}$
$W^{(1)}$
$b^{(1)}$
$W^{(2)}$
$b^{(2)}$
$W^{(3)}$
Input layer
Hidden_1 layer
Hidden_2 layer
Output layer

Input layer

The neurons, colored in **purple**, represent the input data. These can be as simple as scalars or more complex like vectors or multidimensional matrices.

$$x_i = a_i^{(1)}, i \in 1,2,3,4$$

The first set of activations (a) are equal to the input values. *NB: "activation" is the neuron's value after applying an activation function. See below.*

Hidden layers

L=2

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

$$a^{(2)} = f(z^{(2)})$$

L=3

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$$

$$a^{(3)} = f(z^{(3)})$$

Activations a^2 and a^3 are computed using an activation function f. Typically, this **function f is non-linear** (e.g. sigmoid, ReLU, tanh) and allows the network to learn complex patterns in data.

Hidden Layer

- W^1 is a weight matrix of shape (n, m) where n is the number of output neurons (neurons in the next layer) and m is the number of input neurons (neurons in the previous layer). For us, $n = 2$ and $m = 4$.

$$W^{(1)} = \begin{bmatrix} W_{11}^{(1)} & W_{12}^{(1)} & W_{13}^{(1)} & W_{14}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} & W_{23}^{(1)} & W_{24}^{(1)} \end{bmatrix}$$

- **The first number in any weight's subscript matches the index of the neuron in the next layer** (in our case this is the *Hidden_2 layer*) **and the second number matches the index of the neuron in previous layer** (in our case this is the *Input layer*).

Input LAyer

- x is the input vector of shape $(m, 1)$ where m is the number of input neurons. For us, $m = 4$.

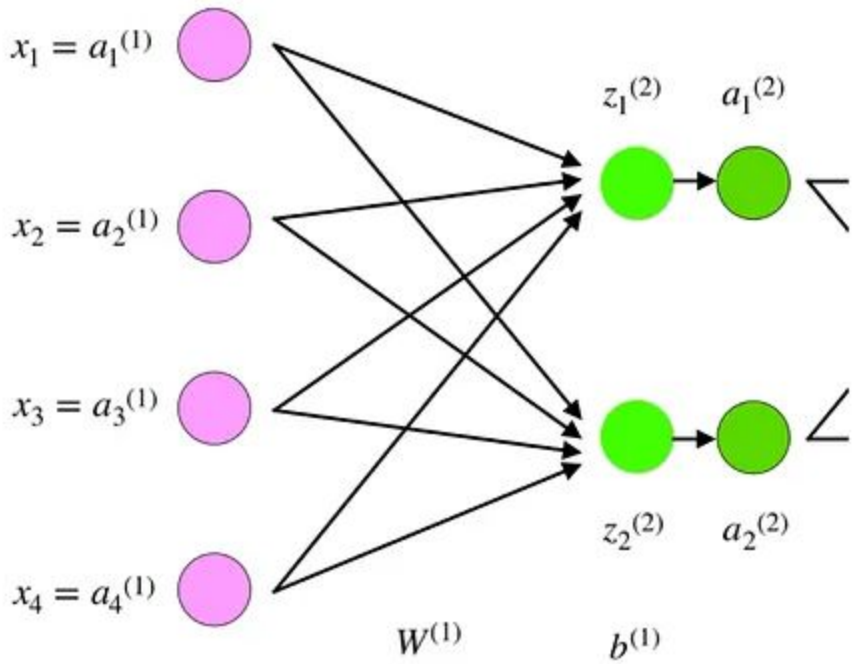
$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

Bias

- b^1 is a bias vector of shape $(n, 1)$ where n is the number of neurons in the current layer.

  For us, $n = 2$.

$$b^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix}$$



$$z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \end{bmatrix}$$

Equation for z²

$$z^{(2)} = \begin{bmatrix} W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + W_{14}^{(1)}x_4 \\ W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + W_{24}^{(1)}x_4 \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix}$$

Output layer

The final part of a neural network is the output layer which produces the predicted value. In our simple example, it is presented as a single neuron, colored in **blue** and evaluated as follows:

$$s = W^{(3)}a^{(3)}$$

Overview of forward propagation equations colored by layer

$$x = a^{(1)} \qquad \textit{Input layer}$$

$$z^{(2)} = W^{(1)}x + b^{(1)} \qquad \textit{neuron value at } Hidden_1 \textit{ layer}$$

$$a^{(2)} = f(z^{(2)}) \qquad \textit{activation value at } Hidden_1 \textit{ layer}$$

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)} \qquad \textit{neuron value at } Hidden_2 \textit{ layer}$$

$$a^{(3)} = f(z^{(3)}) \qquad \textit{activation value at } Hidden_2 \textit{ layer}$$

$$s = W^{(3)}a^{(3)} \qquad \textit{Output layer}$$

- The final step in a forward pass is to evaluate the **predicted output *s*** against an **expected output *y***.
- The output *y* is part of the training dataset *(x, y)* where *x* is the input (as we saw in the previous section). Evaluation between *s* and *y* happens through a **cost function**. This can be as simple as MSE (mean squared error) or more complex like cross-entropy. We name this cost function *C* and denote it as follows:

$$C = cost(s, y)$$

Backpropagation and computing gradients

- Based on *C*'s value, the model "knows" how much to adjust its parameters in order to get closer to the expected output *y*.
- This happens using the backpropagation algorithm.According to the paper from 1989, backpropagation: *repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector*.and *the ability to create useful new features distinguishes back-propagation from earlier, simpler methods...*
-
- In other words, **backpropagation aims to minimize the cost function by adjusting network's weights and biases.** The level of adjustment is determined by the gradients of the cost function with respect to those parameters.

- *Gradient of a function C(x_1, x_2, …, x_m) in point x is a vector of the partial derivatives of C in x.*

$$\frac{\partial C}{\partial x} = [\frac{\partial C}{\partial x_1}, \frac{\partial C}{\partial x_2}, \ldots, \frac{\partial C}{\partial x_m}]$$

- *The derivative of a function C measures the sensitivity to change of the function value (output value) with respect to a change in its argument x (input value). In other words, the derivative tells us the direction C is going.*
- *The gradient shows how much the parameter x needs to change (in positive or negative direction) to minimize C.*

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} \qquad chain\ rule$$

$$z_j^l = \sum_{k=1}^{m} w_{jk}^l a_k^{l-1} + b_j^l \qquad by\ definition$$

$$m\ -\ number\ of\ neurons\ in\ l-1\ layer$$

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1} \qquad by\ differentiation\ (calculating\ derivative)$$

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} a_k^{l-1} \qquad final\ value$$

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} \qquad chain\ rule$$

$$\frac{\partial z_j^l}{\partial b_j^l} = 1 \qquad by\ differentiation\ (calculating\ derivative)$$

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}1 \qquad final\ value$$

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \qquad local\ gradient$$

optimizing weights and biases (also called “Gradient descent”)

while (termination condition not met)

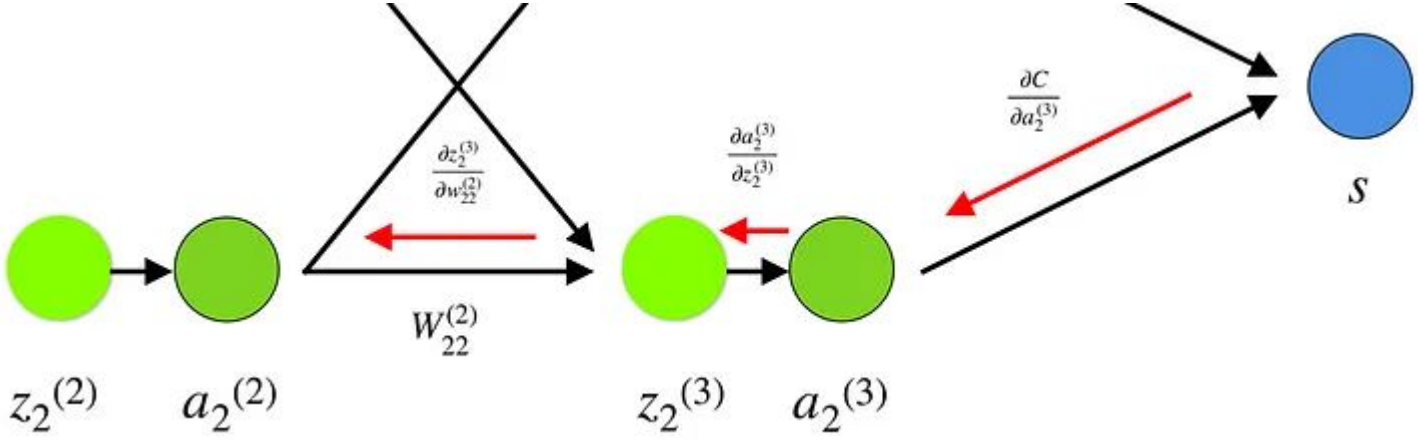
$$w := w - \epsilon \frac{\partial C}{\partial w}$$

$$b := b - \epsilon \frac{\partial C}{\partial b}$$

end

- Initial values of w and b are randomly chosen.
- Epsilon (e) is the learning rate. It determines the gradient's influence.
- w and b are matrix representations of the weights and biases. Derivative of C in w or b can be calculated using partial derivatives of C in the individual weights or biases.
- Termination condition is met once the cost function is minimized.

Visual representation of backpropagation in a neural network



Equation for derivative of C in $(w\_22)^2$

$$\frac{\partial C}{\partial w_{22}^{(2)}} = \frac{\partial C}{\partial z_2^{(3)}} \cdot \frac{\partial z_2^{(3)}}{\partial w_{22}^{(2)}} = \frac{\partial C}{\partial a_2^{(3)}} \cdot \frac{\partial a_2^{(3)}}{\partial z_2^{(3)}} \cdot a_2^{(2)} = \frac{\partial C}{\partial a_2^{(3)}} \cdot f'(z_2^{(3)}) \cdot a_2^{(2)}$$