

2 Machine learning Approaches

Machine learning is closely related to computational statistics, which also focuses in prediction-making through the use of computers. It has strong ties to mathematical optimization, which delivers methods, theory and application domains to the field. Machine learning is sometimes conflated with data mining, where the latter subfield focuses more on exploratory data analysis and is known as unsupervised learning. Machine learning can also be unsupervised and be used to learn and establish baseline behavioral profiles for various entities and then used to find meaningful anomalies.

The most common approaches of machine learning are:

1. Artificial neural networks (Error Back Propagation Neural Network).
2. Clustering (K-mean clustering Algorithm, K-nearest neighbor).
3. Decision tree learning (ID3, C4.5)
4. Bayesian networks.
5. Support vector machines.

2.1 Artificial neural networks:

An artificial neural network (ANN) learning algorithm, usually called "neural network" (NN), is a learning algorithm that is inspired by the structure and functional aspects of biological neural networks. Computations are structured in terms of an interconnected group of artificial neurons, processing information using a connectionist approach to computation. Modern neural networks are non-linear statistical data modeling tools. They are usually used to model complex relationships between inputs and outputs, to find patterns in data, or to capture the statistical structure in an unknown joint probability distribution between observed variables.

Error Back Propagation Neural Network Algorithm:

Artificial neural networks are statistical learning models, inspired by biological neural networks (central nervous systems, such as the brain), that are used in machine learning. These networks are represented as systems of interconnected "**neurons**", which send messages to each other. The connections within the network can be systematically adjusted based on inputs and outputs, making them ideal for supervised learning.

A neural network is a collection of "**neurons**" with "**synapses**" connecting them. The collection is organized into three main parts: the **input layer**, the **hidden layer**, and the **output layer**. Note that you can have n hidden layers, with the term "**deep**" learning implying multiple hidden layers.

Hidden layers are necessary when the neural network has to make sense of something really complicated, contextual, or non-obvious, like image recognition. The term "**deep**" learning came from having **many hidden layers**. These layers are known as "hidden", since they are not visible as a network output.

The **circles represent neurons** and **lines represent synapses**. Synapses take the **input** and multiply it by a "**weight**" (the strength of the input in determining the output). Neurons add the outputs from all synapses and apply an activation function.

Training a neural network basically means calibrating all of the "weights" by repeating two key steps, **forward propagation** and **back propagation**.

Since neural networks are great for regression, the best input data are numbers (as opposed to discrete values, like colors or movie genres, whose data is better for statistical classification models). The output data will be a number within a range like 0 and 1 (this ultimately depends on the activation function).

In **forward propagation**, we apply a set of weights to the input data and calculate an output. For the first forward propagation, the **set of weights is selected randomly**.

In **back propagation**, we measure the margin of error of the output and adjust the weights accordingly to decrease the error.

Neural networks repeat both forward and back propagation until the weights are calibrated to accurately predict an output.

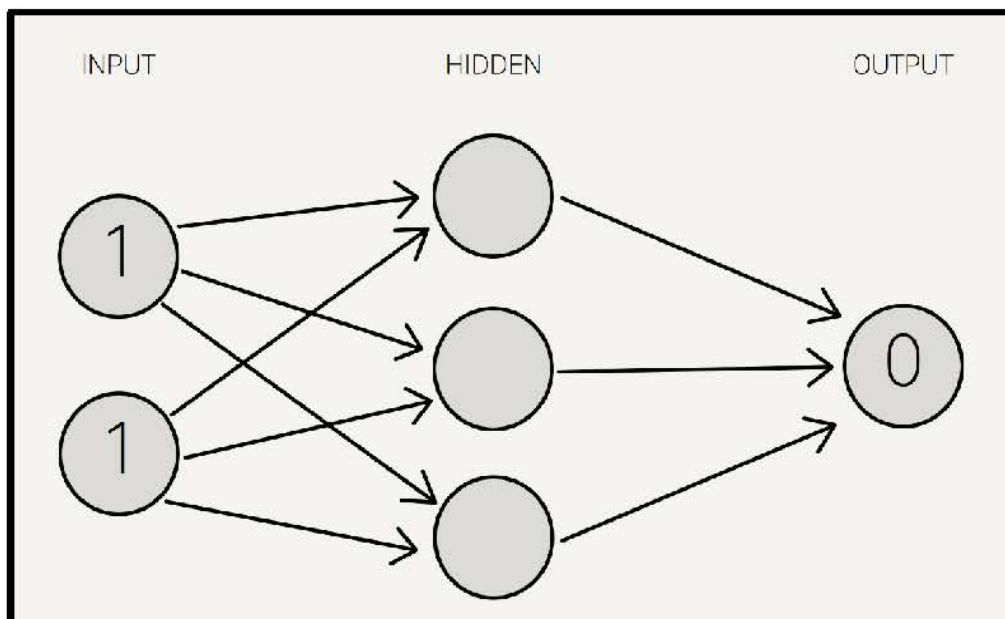
Example: Training a neural network to function as an "Exclusive OR" ("XOR") to illustrate each step in the training process.

Forward Propagation

The XOR function can be represented by the mapping of the below inputs and outputs, which we'll use as training data. It should provide a correct output given any input acceptable by the XOR function.

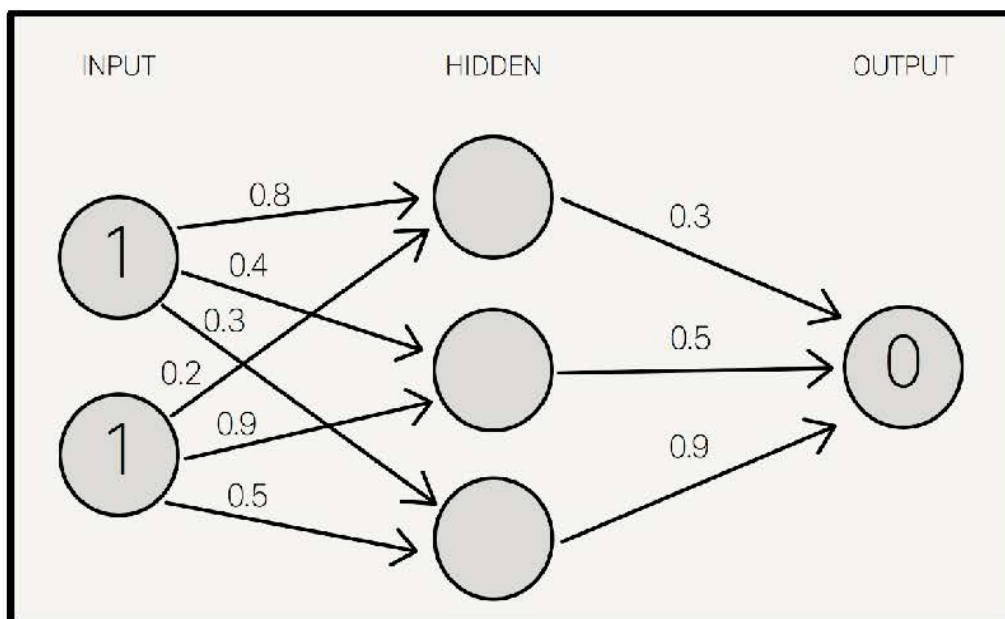
input	output
0, 0	0
0, 1	1
1, 0	1
1, 1	0

Let's use the last row from the above table, $(1, 1) \Rightarrow 0$, to demonstrate forward propagation:



Note that we use a single hidden layer with only three neurons for this example.

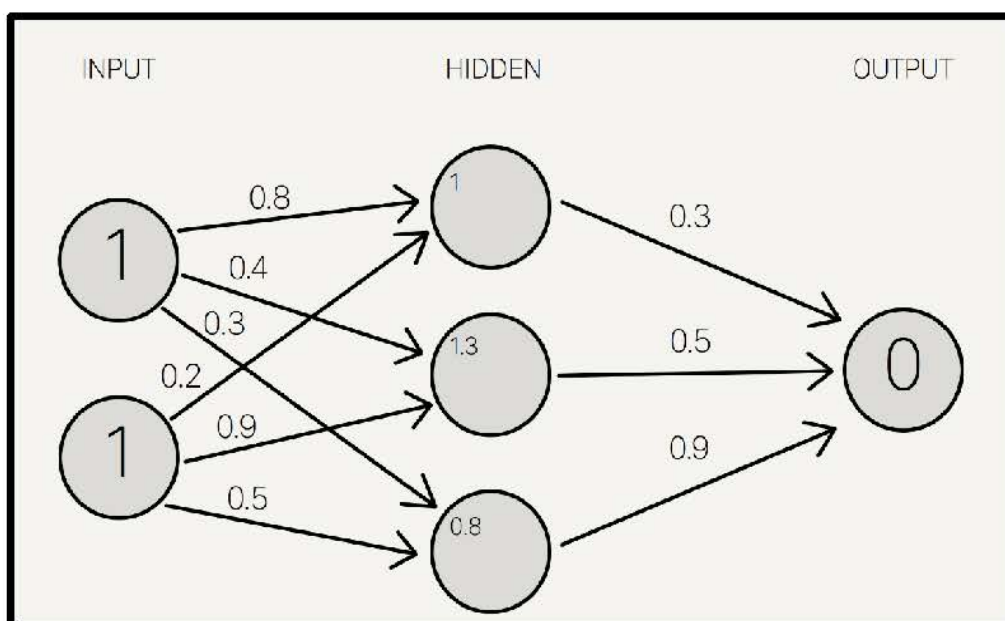
We now assign weights to all of the synapses. Note that these weights are selected randomly (based on Gaussian distribution) since it is the first time we're forward propagating. The initial weights will be between 0 and 1, but note that the final weights don't need to be.



We sum the product of the inputs with their corresponding set of weights to arrive at the first values for the hidden layer. You can think of the weights as measures of influence the input nodes have on the output.

$$\begin{aligned}1 * 0.8 + 1 * 0.2 &= 1 \\1 * 0.4 + 1 * 0.9 &= 1.3 \\1 * 0.3 + 1 * 0.5 &= 0.8\end{aligned}$$

We put these sums smaller in the circle, because they're not the final value:

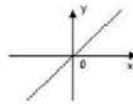
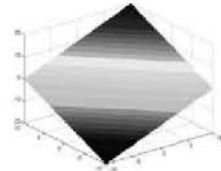
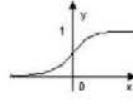
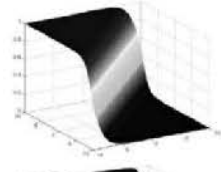
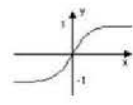
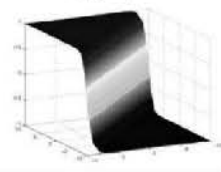


To get the final value, we apply the **activation function** to the hidden layer sums.

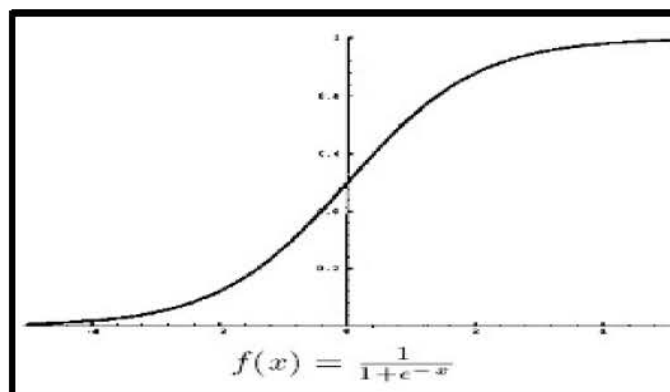
The purpose of the activation function is to transform the input signal into an output signal and are necessary for neural networks to model complex non-linear patterns that simpler models might miss.

There are many types of activation functions:

- **Linear.**
- **Sigmoid.**
- **hyperbolic tangent**

Activation Function	Mathematical Equation	2D Graphical Representation	3D Graphical Representation
Linear	$y = x$		
Sigmoid (logistic)	$y = \frac{1}{1 + e^{-x}}$		
Hyperbolic tangent	$y = \frac{1 - e^{-2x}}{1 + e^{2x}}$		

For our example, let's use the **sigmoid function** for activation. The sigmoid function looks like this, graphically:



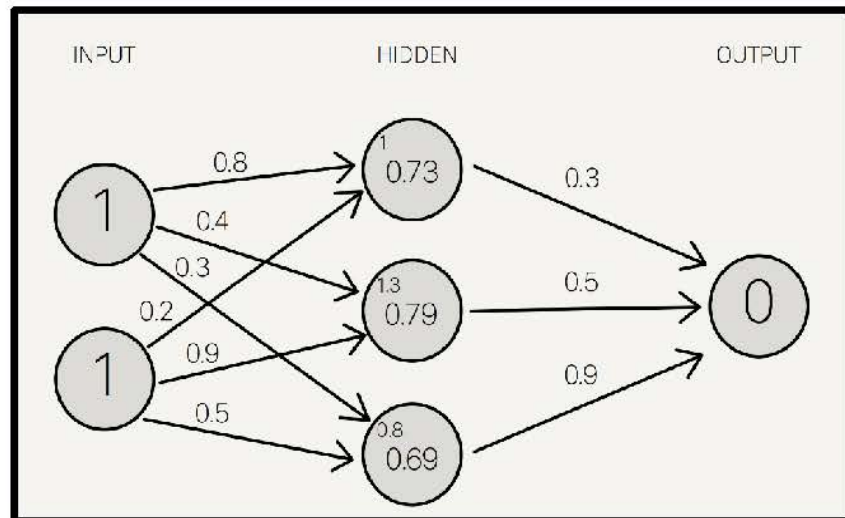
And applying $S(x)$ to the three hidden layer *sums*, we get:

$$S(1.0) = 0.73105857863$$

$$S(1.3) = 0.78583498304$$

$$S(0.8) = 0.68997448112$$

We add that to our neural network as hidden layer *results*:



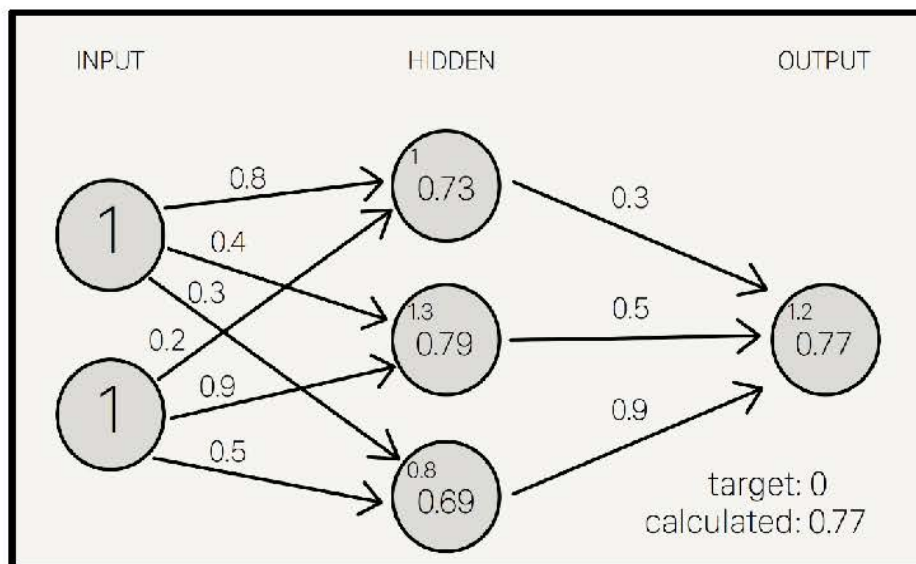
Then, we sum the product of the hidden layer results with the second set of weights (also determined at random the first time around) to determine the output sum.

$$0.73 * 0.3 + 0.79 * 0.5 + 0.69 * 0.9 = 1.235$$

Finally, we apply the activation function to get the final output result.

$$S(1.235) = 0.7746924929149283$$

This is our full diagram:

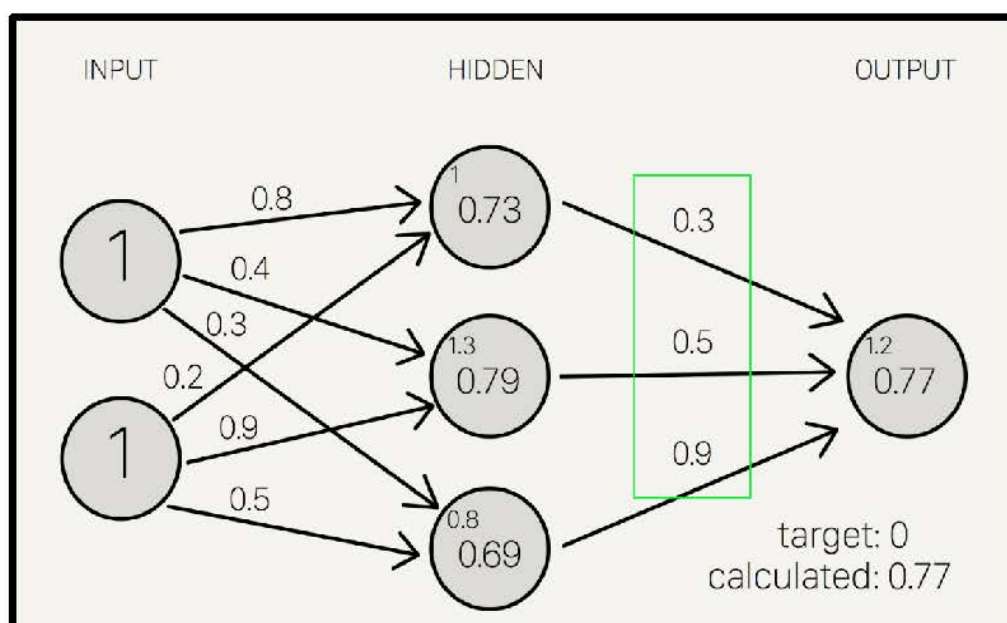


Because we used a random set of initial weights, the value of the output neuron is off the mark; in this case by +0.77 (since the target is 0). If we stopped here, this set of weights would be a great neural network for inaccurately representing the XOR operation.

Back Propagation

To improve our model, we first have to quantify just how wrong our predictions are. Then, we adjust the weights accordingly so that the margin of errors is decreased.

Similar to forward propagation, back propagation calculations occur at each "layer". We begin by changing the weights between the hidden layer and the output layer.



Calculating the incremental change to these weights by two steps:

- 1- **Find the margin of error of the output result** (what we get after applying the activation function) to back out the necessary change in the output sum (we call this delta output sum) and
- 2- **Extract the change in weights** by multiplying delta output sum by the hidden layer results.

The output sum margin of error is the target output result minus the calculated output result:

Output Sum Margin of Error = Target - Calculated

Target = 0

Calculated = 0.77

Target - calculated = -0.77

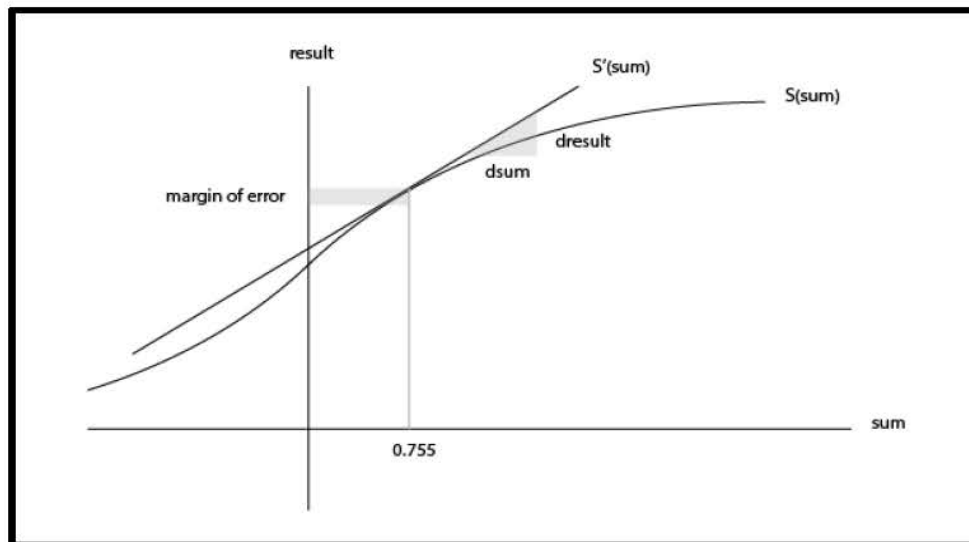
To calculate the necessary change in the output sum, or delta output sum, we take the derivative of the activation function and apply it to the output sum. In our example, the activation function is the sigmoid function.

$$\text{Delta output sum} = S'(\text{output sum}) * (\text{output sum margin of error})$$

$$\text{Delta output sum} = S'(1.235) * (-0.77)$$

$$\text{Delta output sum} = -0.13439890643886018$$

Here is a graph of the Sigmoid function to give you an idea of how we are using the derivative to move the input towards the right direction. Note that this graph is not to scale.



Now that we have the proposed change in the output layer sum (-0.13), let's use this in the derivative of the output sum function to determine the new change in weights.

As a reminder, the mathematical definition of the output sum is the product of the hidden layer result and the weights between the hidden and output layer:

$$H_{result} \times w_{h \rightarrow o} = O_{sum}$$

$$\text{hidden result 1} = 0.73105857863$$

$$\text{hidden result 2} = 0.78583498304$$

$$\text{hidden result 3} = 0.68997448112$$

$$\text{Delta weights} = \text{delta output sum} * \text{hidden layer results}$$

$$\text{Delta weights} = -0.1344 * [0.73105, 0.78583, 0.69997]$$

$$\text{Delta weights} = [-0.0983, -0.1056, -0.0941]$$

$$\text{old } w_7 = 0.3$$

$$\text{old } w_8 = 0.5$$

$$\text{old } w_9 = 0.9$$

new w7 = 0.202

new w8 = 0.394

new w9 = 0.806

To determine the change in the weights between the input and hidden layers, we perform the similar, but notably different, set of calculations. Note that in the following calculations, we use the initial weights instead of the recently adjusted weights from the first part of the backward propagation.

We can determine the delta hidden sum:

Delta hidden sum = delta output sum * hidden-to-output weights * S'(hidden sum)

Delta hidden sum = -0.1344 * [0.3, 0.5, 0.9] * S'([1, 1.3, 0.8])

Delta hidden sum = [-0.0403, -0.0672, -0.1209] * [0.1966, 0.1683, 0.2139]

Delta hidden sum = [-0.0079, -0.0113, -0.0259]

Once we get the delta hidden sum, we calculate the change in weights between the input and hidden layer by multiplying it by the input data, (1, 1). The input data here is equivalent to the hidden results in the earlier back propagation process to determine the change in the hidden-to-output weights.

Let's do the math:

input 1 = 1

input 2 = 1

Delta weights = delta hidden sum * input

Delta weights = [-0.0079, -0.0113, -0.0259] * [1, 1]

Delta weights = [-0.0079, -0.0113, -0.0259, -0.0079, -0.0113, -0.0259]

old w1 = 0.8

old w2 = 0.4

old w3 = 0.3

old w4 = 0.2

old w5 = 0.9

old w6 = 0.5

new w1 = 0.7921

new w2 = 0.3887

new w3 = 0.2741

new w4 = 0.1921

new w5 = 0.8887

new w6 = 0.4741

Here are the new weights, right next to the initial random starting weights as comparison:

<u>old</u>	<u>new</u>
w1: 0.8	w1: 0.7921
w2: 0.4	w2: 0.3887
w3: 0.3	w3: 0.2741
w4: 0.2	w4: 0.1921
w5: 0.9	w5: 0.8887
w6: 0.5	w6: 0.4741
w7: 0.3	w7: 0.2020
w8: 0.5	w8: 0.3940
w9: 0.9	w9: 0.8060

Once we arrive at the adjusted weights, we start again with forward propagation. When training a neural network, it is common to repeat both these processes thousands of times (by default, Mind iterates 10,000 times). And doing a quick forward propagation with our new weights, we can see that the final output is now 0.73, which is just a little bit closer to the expected output. Through just one iteration of forward and back propagation, we've already improved the network!