Introduction to Artificial Neural Networks - Part 1 (http://www.theprojectspot.com/)

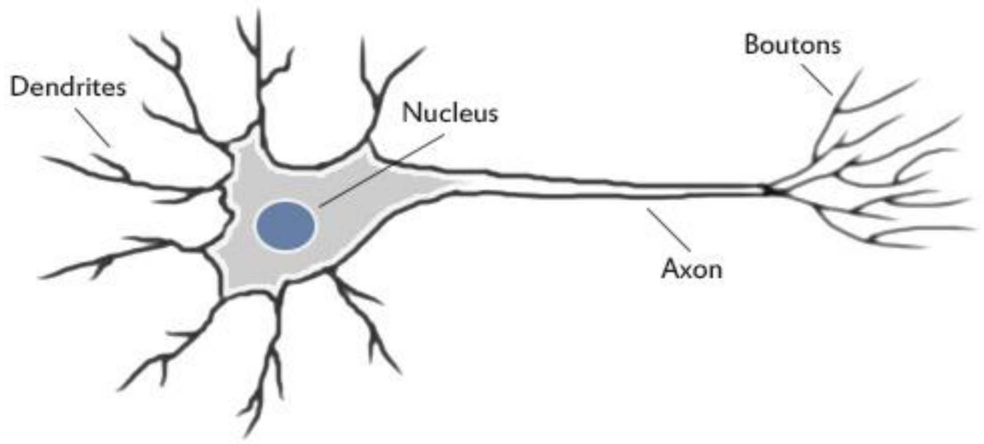
ARTIFICIAL NEURAL NETWORK

Introduction

Computers are great at solving algorithmic and math problems, but often the world can't easily be defined with a mathematical algorithm. Facial recognition and language processing are a couple of examples of problems that can't easily be quantified into an algorithm, however these tasks are trivial to humans. The key to Artificial Neural Networks is that their design enables them to process information in a similar way to our own biological brains, by drawing inspiration from how our own nervous system functions. This makes them useful tools for solving problems like facial recognition, which our biological brains can do easily.

How do they work?

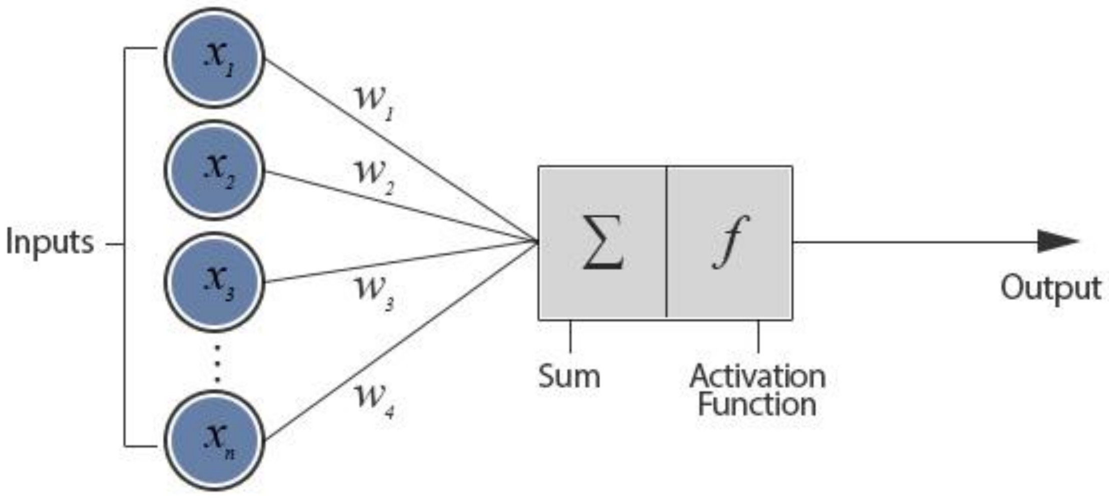
First lets take a look at what a biological neuron looks like.



Our brains use extremely large interconnected networks of neurons to process information and model the world we live in. Electrical inputs are passed through this network of neurons which result in an output being produced. In the case of a biological brain this could result in contracting a muscle or signaling your sweat glands to produce sweat. A neuron collects inputs using a structure called dendrites, the neuron effectively sums all of these inputs from the dendrites and if the resulting value is greater than it's firing threshold, the neuron fires. When the neuron fires it sends an electrical impulse through the neuron's axon to it's boutons. These boutons can then be networked to thousands of other neurons via connections called synapses. There are about one hundred billion (100,000,000,000) neurons inside the human brain each with about one thousand synaptic connections. It's effectively the way in which these synapses are wired that give our brains the ability to process information the way they do.

Modeling Artificial Neurons

Artificial neuron models are at their core simplified models based on biological neurons. This allows them to capture the essence of how a biological neuron functions. We usually refer to these artificial neurons as 'perceptrons'. So now lets take a look at what a perceptron looks like.



As shown in the diagram above a typical perceptron will have many inputs and these inputs are all individually weighted. The perceptron weights can either amplify or deamplify the original input signal. For example, if the input is 1 and the input's weight is 0.2 the input will be decreased to 0.2. These weighted signals are then added together and passed into the activation function. The activation function is used to convert the input into a more useful output. There are many different types of activation function but one of the simplest would be step function. A step function will typically output a 1 if the input is higher than a certain threshold, otherwise it's output will be 0.

Here's an example of how this might work:

Input 1 (x_1) = 0.6

Input 2 (x_2) = 1.0

Weight 1 (w_1) = 0.5

Weight 2 (w_2) = 0.8

Threshold = 1.0

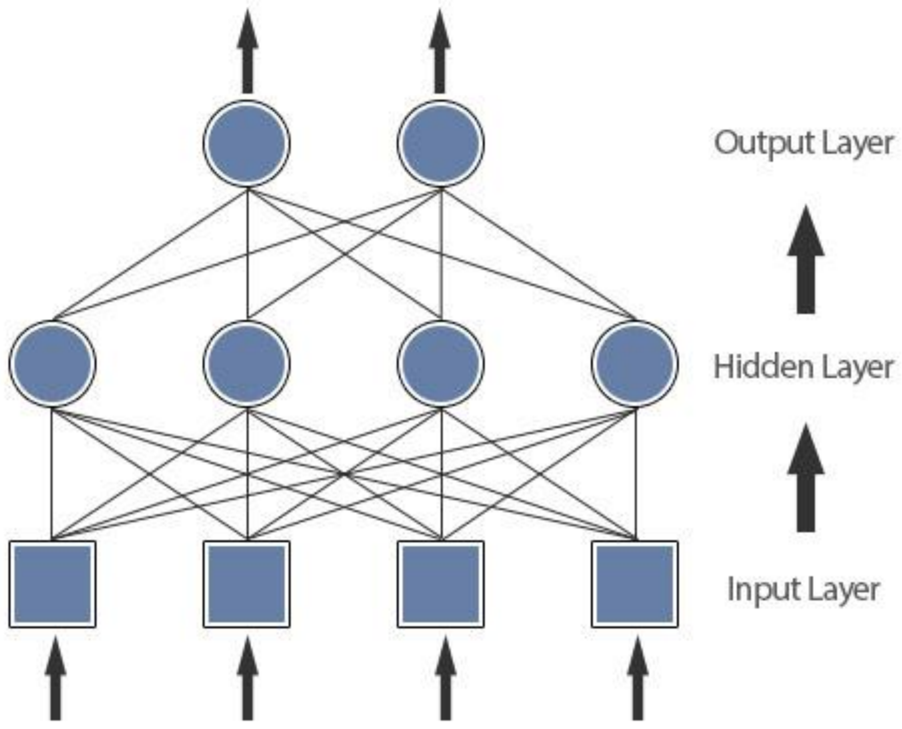
First we multiple the inputs by their weights and sum them:

$x_1w_1 + x_2w_2 = (0.6 \times 0.5) + (1 \times 0.8) = 1.1$

Now we compare our input total to the perceptron's activation threshold. In this example the total input (1.1) is higher than the activation threshold (1.0) so the neuron would fire.

Implementing Artificial Neural Networks

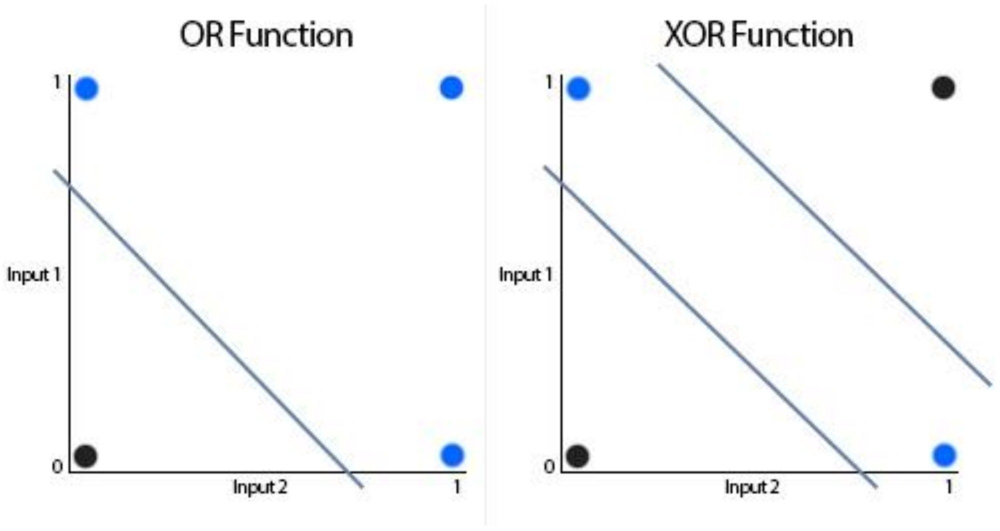
So now you're probably wondering what an artificial neural network looks like and how it uses these artificial neurons to process information. In this tutorial we're going to be looking at feedforward networks and how their design links our perceptron together creating a functioning artificial neural network. Before we begin lets take a look at what a basic feedforward network looks like:



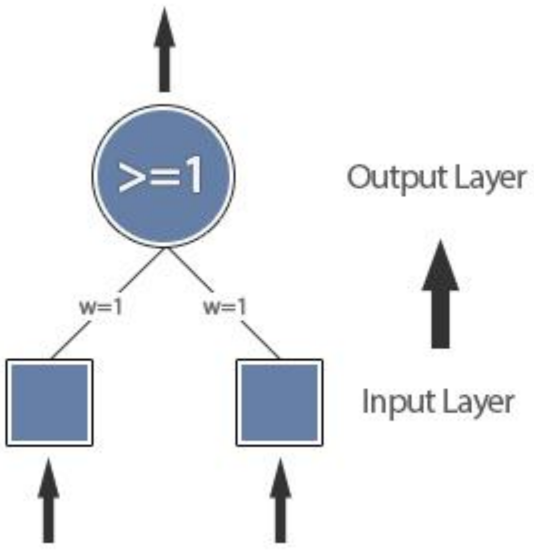
Each input from the input layer is fed up to each node in the hidden layer, and from there to each node on the output layer. We should note that there can be any number of nodes per layer and there are usually multiple hidden layers to pass through before ultimately reaching the output layer. Choosing the right number of nodes and layers is important later on when optimizing the neural network to work well a given problem. As you can probably tell from the diagram, it's called a feedforward network because of how the signals are passed through the layers of the neural network in a single direction. These aren't the only type of neural network though. There are also feedback networks where its architecture allows signals to travel in both directions.

Linear separability

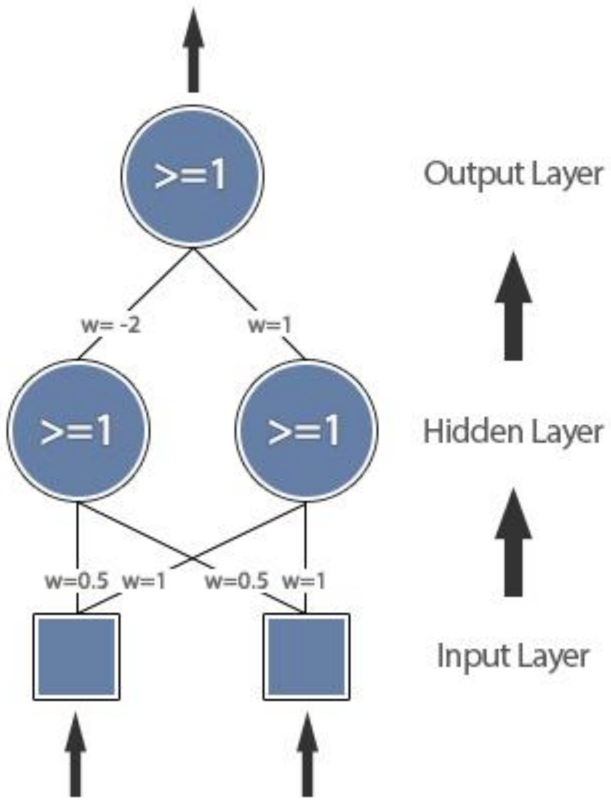
To explain why we usually require a hidden layer to solve our problem, take a look at the following examples:



Notice how the OR function can be separated on the graph with a single straight line, this means the function is "linearly separable" and can be modelled within our neural network without implementing a hidden layer, for example, the OR function can be modeled with a single perceptron like this:



However to model the XOR function we need to use an extra layer:



We call this type of neural network a 'multi-layer perceptron'. In almost every case you should only ever need to use one or two hidden layers, however it make take more experimentation to find the optimal amount of nodes for the hidden layer(s).

Implementing Supervised Learning

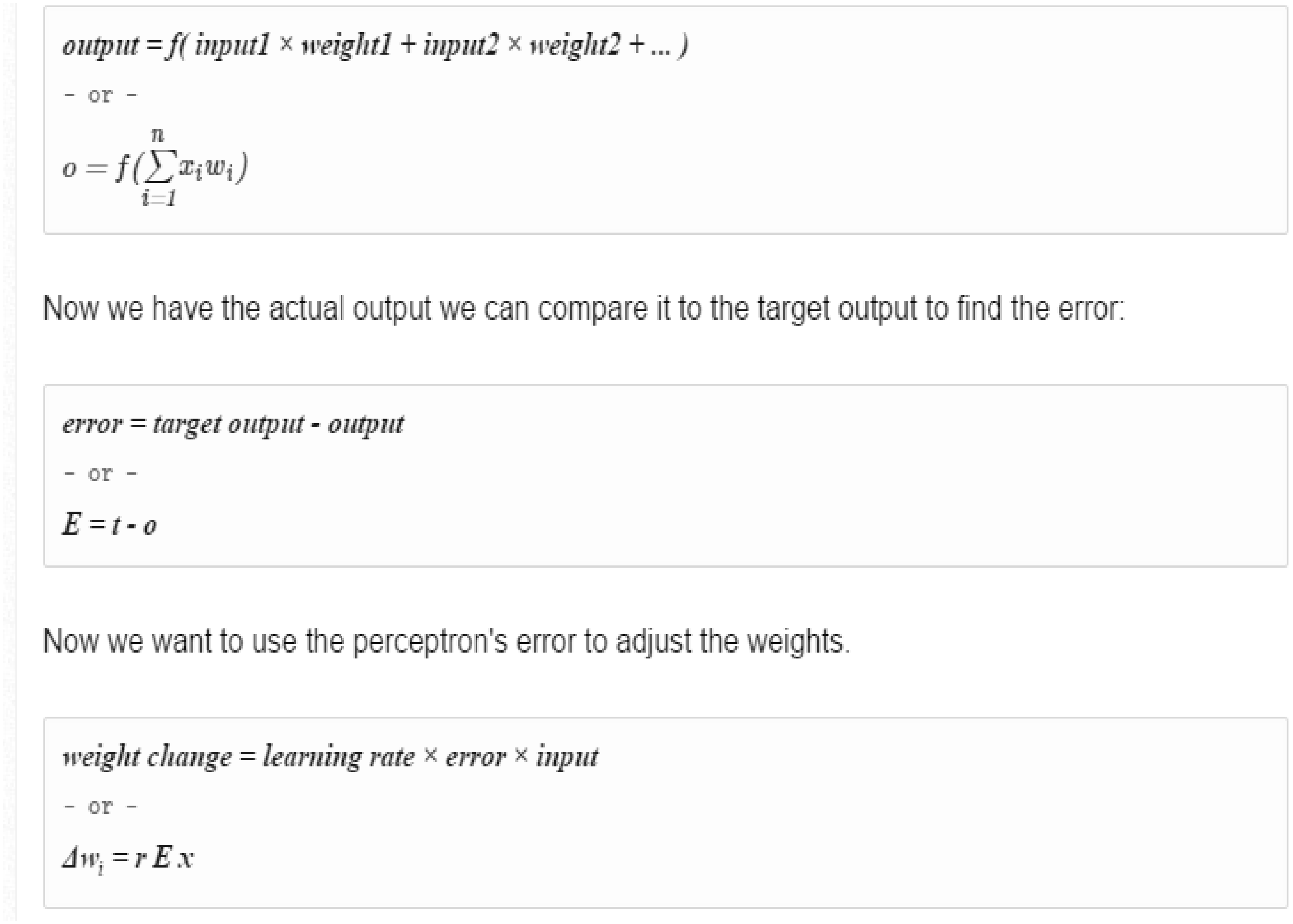
As mentioned earlier, supervised learning is a technique that uses a set of input-output pairs to train the network. The idea to provide the network with examples of inputs and outputs then to let it find a function that can correctly map the data we provided to a correct output. If the network has been trained with a good range of training data when the network has finished learning we should even be able to give it a new, unseen input and the network should be able to map it correctly to an output.

There are many different supervised learning algorithms we could use but the most popular, and the one we will be looking at in more detail is backpropagation. Before we look at why backpropagation is needed to train multi-layered networks, let's first look at how we can train single-layer networks, or as they're otherwise known, perceptron.

The Perceptron Learning rule

The perceptron learning rule works by finding out what went wrong in the network and making slight corrections to hopefully prevent the same errors happening again. Here's how it works... First we take the network's actual output and compare it to the target output in our training set. If the network's actual output and target output don't match we know something went wrong and we can update the weights based on the amount of error. Let’s run through the algorithm step by step to understand how exactly it works.

First, we need to calculate the perceptron's output for each output node. As you should remember from the previous tutorial we can do this by:



$output = f(\,input1 \times weight1 + input2 \times weight2 + \ldots\,)$

- or -

$o = f(\sum_{i=1}^{n} x_i w_i)$

Now we have the actual output we can compare it to the target output to find the error:

$error = target\ output - output$

- or -

$E = t - o$

Now we want to use the perceptron's error to adjust the weights.

$weight\ change = learning\ rate \times error \times input$

- or -

$\Delta w_i = r\,E\,x$

We want to ensure only small changes are made to the weights on each iteration, so to do this we apply a small learning rate (r). If the learning rate is too high the perceptron can jump too far and miss the solution, if it's too low, it can take an unreasonably long time to train.

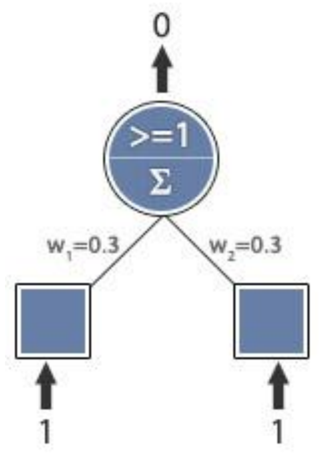
This gives us a final weight update equation of:

weight change = learning rate × (target output - actual output) × input

- or -

$\Delta w_i = r (t - o) x_i$

Here's an example of how this would work with the AND function...



Learning rate = 0.1

Expected output = 1

Actual output = 0

Error = 1

Weight Update:

$w_i = r E x + w_i$

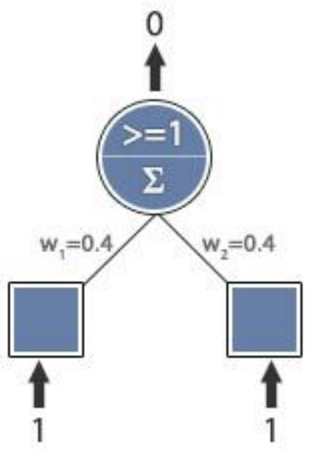
$w_1 = 0.1 \times 1 \times 1 + w_1$

$w_2 = 0.1 \times 1 \times 1 + w_2$

New Weights:

$w_1 = 0.4$

$w_2 = 0.4$



Learning rate = 0.1

Expected output = 1

Actual output = 0

Error = 1

Weight Update:

$w_i = r\ E\ x + w_i$

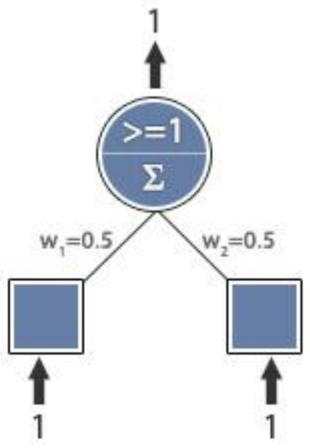
$w_1 = 0.1 \times 1 \times 1 + w_1$

———————————————$w_2 = 0.1 \times 1 \times 1 + w_2$

New Weights:

$w_1 = 0.5$

$w_2 = 0.5$



Learning rate = 0.1

Expected output = 1

Actual output = 1

Error = 0

No error,

training complete.