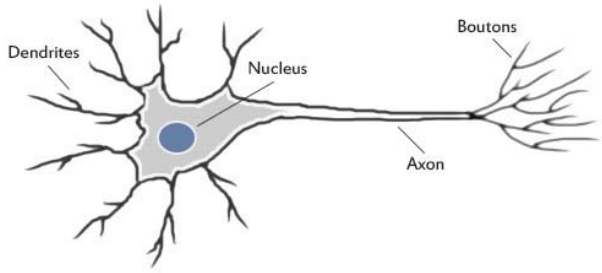
(Parts 1 & 2) For Dummies—
The Introduction to Neural Networks we all need!

Harsh PokharnaFollow
Blockchains | Mobile Application Development | IIT Kanpur Alum | Ex-Flipkart | Ex-JioMoney | Classic Rock | Guitar | Fitness | Long Drives
Jul 26, 2016

This is going to be a 2 article series. This article gives an introduction to perceptrons (single layered neural networks)

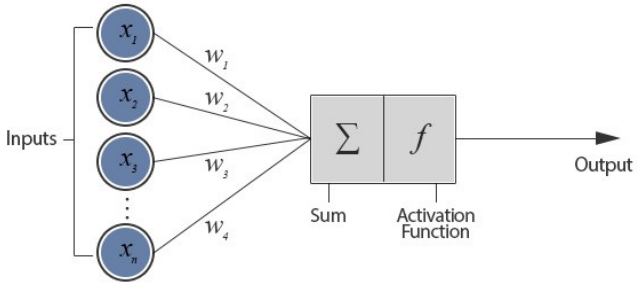
***Update**: Part2 of the series is now available for reading here!*



A neuron in our brain

Our brain uses the extremely large interconnected network of neurons for information processing and to model the world around us. Simply put, a neuron collects inputs from other neurons using dendrites. The neuron sums all the inputs and if the resulting value is greater than a threshold, it fires. The fired signal is then sent to other connected neurons through the axon.

Now, how do we model artificial neurons?



Model of an artificial neuron

The figure depicts a neuron connected with *n* other neurons and thus receives *n* inputs (x1, x2, ….. xn). This configuration is called a **Perceptron.**

The inputs (x1, x2, …. xn) and weights (w1, w2, …. wn) are real numbers and can be positive or negative.

The perceptron consists of weights, summation processor and an activation function.

Note*: It also contains a threshold processor (known as bias) but we will talk about that later!*

All the inputs are individually weighted, added together and passed into the activation function. 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.

Note: *There are other activation functions too such as sigmoid, etc which are used in practice.*

An example would be,

Input 1 (x1) = 0.6
Input 2 (x2) = 1.0

Weight 1 (w1) = 0.5
Weight 2 (w2) = 0.8

Threshold = 1.0

Weighing the inputs and adding them together gives,

x1w1 + x2w2 = (0.6 x 0.5) + (1 x 0.8) = 1.1

Here, the total input is higher than the threshold and thus the neuron fires.

Training in perceptrons!

Try teaching a child to recognize a bus? You show her examples, telling her, "This is a bus. That is not a bus," until the child learns the concept of what a bus is. Furthermore, if the child sees new objects that she hasn't seen before, we could expect her to recognize correctly whether the new object is a bus or not.

This is exactly the idea behind the perceptron.

Similarly, input vectors from a training set are presented to the perceptron one after the other and weights are modified according to the following equation,

For all inputs i,

W(i) = W(i) + a*(T-A)*P(i), where a is the learning rate

Note: Actually the equation is
W(i) = W(i) + a*g'(sum of all inputs)*(T-A)*P(i), where g' is the derivative of the activation function.
Since it is problematic to deal with the derivative of step function, we drop that out of the equation here.

Here, W is the weight vector. P is the input vector. T is the correct output that the perceptron should have known and A is the output given by the perceptron.

When an entire pass through all of the input training vectors is completed without an error, the perceptron has learnt!

At this time, is an input vector P (already in the training set) is given to the perceptron, it will output the correct value. If P is not in the training set, the network will respond with an output similar to other training vectors close to P.

What is the perceptron actually doing?

The perceptron is adding all the inputs and separating them into 2 categories, those that cause it to fire and those that don't. That is, it is drawing the line:

w1x1 + w2x2 = t, where t is the threshold

and looking at where the input point lies. Points on one side of the line fall into 1 category, points on the other side fall into the other category. And because the weights and thresholds can be anything, this is just *any line* across the 2 dimensional input space.

Limitation of Perceptrons

Not every set of inputs can be divided by a line like this. Those that can be are called *linearly separable*. If the vectors are not linearly separable, learning will never reach a point where all vectors are classified properly. The most famous example of the perceptron's inability to solve problems with linearly non-separable vectors is the boolean XOR problem.

The next part of this article series will show how to do this using **muti-layer neural networks**, using the back propagation training method.

If you enjoyed reading this article, hit the little green heart button to show your love!

To stay updated for the next article in the series, please follow :)

And if you want your friends to read this too, click share!

References: http://www.codeproject.com/Articles/16508/AI-Neural-Network-for-beginners-Part-of

http://www.theprojectspot.com/tutorial-post/introduction-to-artificial-neural-networks-part-1/7

https://medium.com/technologymadeeasy/for-dummies-the-introduction-to-neural-networks-we-all-need-c50f6012d5eb

https://medium.com/technologymadeeasy/for-dummies-the-introduction-to-neural-networks-we-all-need-part-2-1218d5dc043

This article is in continuation to the Part1 of this series. If you have not yet read it, I highly recommend you to do that before we dive into multi layered neural networks here!

Just as a recap, I will quickly go through what a single layered neural network basically does. Once a training sample is feeded to the network, each output node of the single layered neural network (also called **Perceptron**) takes a weighted sum of all the inputs and pass them through an activation function (probably sigmoid or step) and comes up with an output. The weights are then corrected using the following equation,

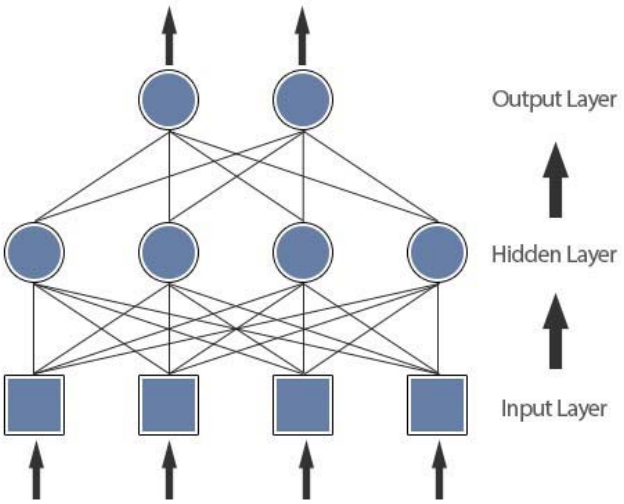
For all inputs i,

W(i) = W(i) + a*g'(sum of all inputs)*(T-A)*P(i), where a is the learning rate and g' is the derivative of the activation function.

Note: *We drop the derivative function in case the activation function is a step function*.

This process is repeated by feeding the whole training set several times until the network responds with a correct output for all the samples. The training is possible only for inputs that are linearly separable. This is where multi-layered neural networks come into picture.

What are multi-layered neural networks?



A multi-layered neural network

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.

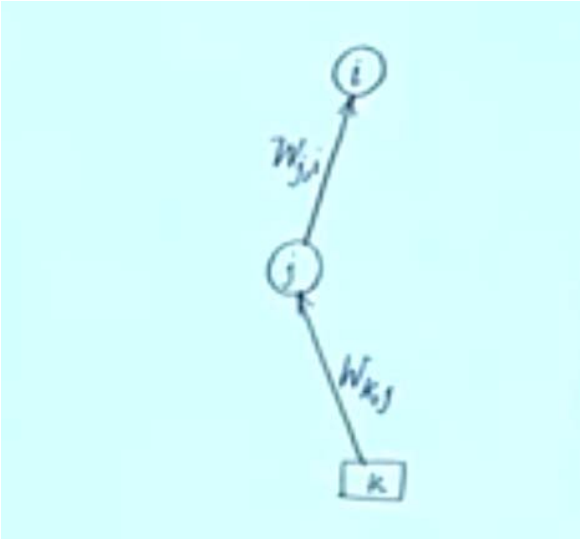
But to train this network we need a learning algorithm which should be able to tune **not only** the weights between the output layer and the hidden layer **but also** the weights between the hidden layer and the input layer.

Enters Back Propagation!

First of all, we need to understand what do we lack. To tune the weights between the hidden layer and the input layer, we need to know the error at the hidden layer, but we know the error only at the output layer *(We know the correct output from the training sample and we also know the output predicted by the network.)*

So, the method that was suggested was to take the errors at the output layer and proportionally propagate them backwards to the hidden layer.

Below we will write equation for a 2 layered network but the same concept applies to a network with any number of layers.



One segment of a 2 layered network

We will follow the nomenclature as shown in the above figure.

https://medium.com/technologymadeeasy/for-dummies-the-introduction-to-neural-networks-we-all-need-c50f6012d5eb

https://medium.com/technologymadeeasy/for-dummies-the-introduction-to-neural-networks-we-all-need-part-2-1218d5dc043

For a particular neuron in output layer

for all j{

Wj,i = Wj,i + a*g'(sum of all inputs)*(T-A)*P(j)

}

This equation tunes the weights between the output layer and the hidden layer.

For a particular neuron *j* in hidden layer, we propogate the error backwards from the output layer, thus

Error = Wj,1 * E1 + Wj,2 * E2 + ….. for all the neurons in output layer

Thus,

For a particular neuron in hidden layer

for all k{

Wk,j = Wk,j + a*g'(sum of all inputs)*(T-A)*P(k)

}

This equation tunes the weights between the hidden layer and the input layer.

So, in a nutshell what we are doing is

- We present a training sample to the neural network (initialised with random weights)
- Compute the output received by calculating activations of each layer and thus calculate the error
- Having calculated the error, we readjust the weights (according to the above mentioned equations) such that the error decreases
- We continue the process for all training samples several times until the weights are not changing too much

If you enjoyed reading this article, hit the little green heart button to show your love!

To stay updated with new technologies, please follow :)

And if you want your friends to read this too, click share!

References: http://www.codeproject.com/Articles/16508/AI-Neural-Network-for-beginners-Part-of

http://www.theprojectspot.com/tutorial-post/introduction-to-artificial-neural-networks-part-1/7

https://medium.com/technologymadeeasy/for-dummies-the-introduction-to-neural-networks-we-all-need-c50f6012d5eb

https://medium.com/technologymadeeasy/for-dummies-the-introduction-to-neural-networks-we-all-need-part-2-1218d5dc043