Neural Networks

Graphical Representation of Linear Equations

Consider the following graph:

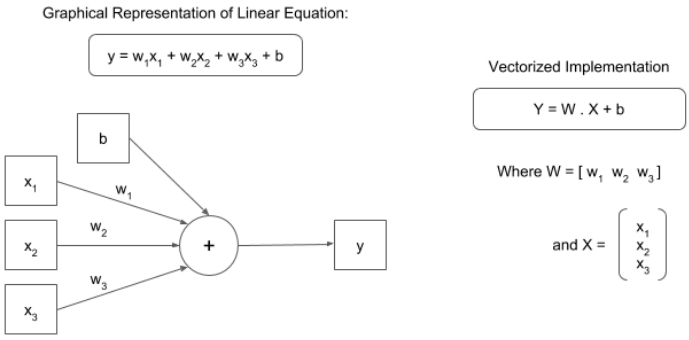

Figure 1: Single Neuron

The above graph simply represents the linear equation:

$$y = w1 * x1 + w2 * x2 + w3 * x3 + b \tag{1}$$

Where the `w1, w2, w3` are called the weights and `b` is an intercept term called bias. The graph above, therefore, is simply a graphical representation of a simple linear equation. The equation can also be *vectorised* like this:

$$y = W.X + b \tag{2}$$

Where `X = [x1, x2, x3]` and `W = [w1, w2, w3].T`. The .T means *transpose*. This is because we want the dot product to give us the result we want i.e. `w1 * x1 + w2 * x2 + w3 * x3`. This gives us the vectorized version of our linear equation.

With machine learning, what we are trying to do, essentially, is find out that if we are given a large amount of data (pairs of X and corresponding y), can we write an algorithm to figure out the optimal values of W and b? We need to find a way for our model to find the *optimal* values for W and b and not the absolute values. Absolute values probably don't even exist given the limitations of our mathematical model since we are *assuming* a linear function for a problem where it might be a much more complex one in reality and we don't know what that function is.

By taking the observed data and a proposed model, we want to write an algorithm to learn the values for W and b which best fit the data and ultimately, by doing that, we learn an approximate function which maps the inputs to outputs of our data. This type of algorithm is called an *optimization* algorithm and there are a few different optimization algorithms that are typically used in training neural networks.

Let's say we have a dataset which has examples of shape `(60000, 28, 28)`. The first dimension is simply the number of examples we have, so each example is of the shape `(28, 28)`. If we unroll this 28 by 28 array into a single dimension, it will become a `28 * 28 = 784` dimensional vector. Now, it can probably be modeled somewhat like a linear equation, right? Given features from `x1` to `x784`, we get an output `y`. It could be represented like this:

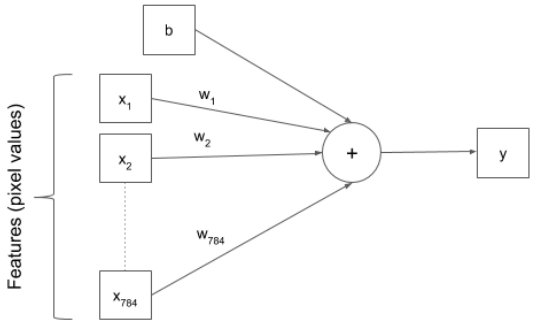

Figure 2: Single Neuron with 784 features

This may actually work for really simple problems but in most cases, this model will turn out to be insufficient. This is where Neural Networks may be more effective.

Neural Network with Two Hidden Layers

Turns out, we can learn much more complex functions by simply *cascading* the linear functions one after the other - and the only additional thing that a node in a neural network does (as opposed to a node in a linear equation shown above), is that an activation function is applied to each linear output. The purpose of an activation functions is to help the neural network find non-linear patterns in the data because if we just cascaded the nodes like the ones described above in the linear equation example, even with many layers of cascaded linear functions, the result will still be a linear function; which means that, after training the mode, it will learn a linear function that best fit the data. This is a problem

because in many, if not most, cases the input to output map is going to be much more complex than a linear function. So, the activation gives the model more flexibility, and allows the model to be able to learn non-linear patterns.

Now, instead of setting y to a weighted sum of our input features, we can get a few hidden outputs which are weighted sums of our input features passed through an activation function and then get the weighted sums of those hidden outputs and so on. We do this a few times, and then get to our output y. This type of model gives our algorithm a much greater chance of learning a complex function.

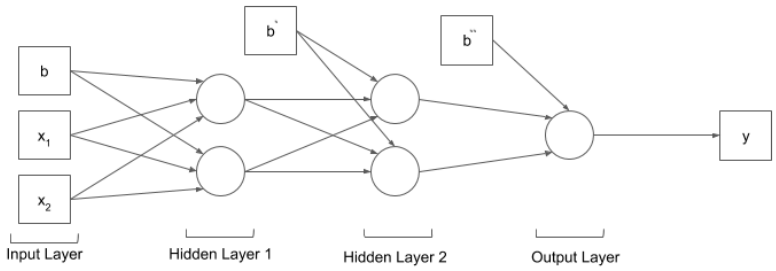

Figure 3: Neural Network with 2 hidden layers

In the network above, we have two *hidden layers*. The first layer with all the X features is called the input layer and the output y is called the output layer. In this example, the output has only one **node**. The hidden layer can have a lot of nodes or a very few nodes depending on how complex the problem may be. Here, both the hidden layer have 2 nodes each. Each node gives the output of a linear function after the linear output passes through an activation function, and takes inputs from each node of the preceding layer. All the W's and all the b's associated with all of these functions will have to be "learned" by our algorithm as it attempts to optimize those values in order to best fit the given data. Note that the total number of learnable parameters in any layer depend on the number of nodes in that layer as well as on the number of nodes in the preceding layer. For example, learnable parameters for **hidden layer 1** can be calculated as: (number of nodes of the layer) * (number of nodes of preceding layer) + (number of nodes of the layer). Why? The first part is obvious: if every node of a layer is connected to every node of the preceding layer, we can simply multiply the number of nodes of these two layers to get the total number of weight parameters. Also, the **bias** from previous layer would be connected to each node in the layer as well - that gives us the second term. So, for **hidden**

layer 1, we get: `2 * 2 + 2 = 6` learnable parameters.

In the hand-written digit classification problem, we will have 128 nodes for two hidden layers, we will have 10 nodes for the output layer with each node corresponding to one output class, and of course we already know that the input is a 784 dimensional vector.

Activation Functions

We have talked about each node having a weighted sum of the inputs of the preceding layer. And, before this sum is fed to the next layer's nodes, it goes through another function called an activation function. So, each node actually does two things. First step is the weighted sum, let's call it Z:

$$Z = W.X + b \tag{3}$$

The second step in the node is the activation function output, let's call it A:

$$A = f(Z) \tag{4}$$

There are various types of activation functions used in Neural Networks. One of the more common ones is a rectified linear unit of ReLU function. It's a pretty simple function: it's a linear function for all the positive values and is simply set to `0` for all the negative values. Something like this:

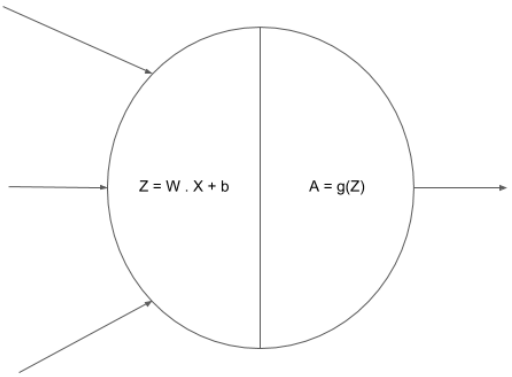

Figure 4: Linear Sum and Activation Function

Another activation function commonly used in classification problems is *softmax*. This function gives us probability scores for various nodes, in this case 10 nodes of the output layer, which sum upto 1. This activation gives us the probabilities

for various classes given the input. The class with the highest probability gives us our prediction.

From Project: Basic Image Classification with TensorFlow by Amit Yadav