The Mathematics of Neural Networks

Opening the ‘black box’

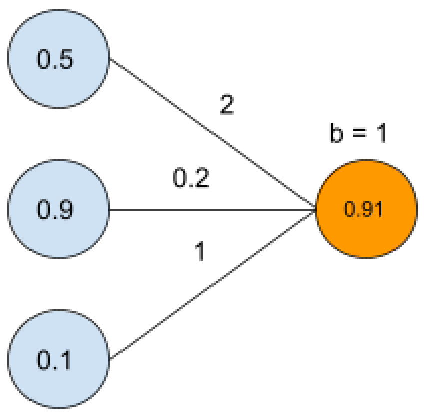
Machine Learning. These two words are suddenly everywhere as discussions of its power and frightening capabilities fill the media and newspapers of the world. We hear every day about these mysterious technologies and seemingly magical algorithms that allow computers to learn how to approach tasks better than any humans. Yet most of us just accept they exist and move on. We collectively choose to treat the word ‘algorithm’ as a black box - as soon as we are told something works based on an algorithm we shrug it off and assume its fantastical mechanisms are too far beyond our intellectual reach to have any hope of comprehending.

In this essay I intend to demystify a crucial machine learning concept - the neural network - and prove that, although these systems rely on reasonably complicated calculus and linear algebra, nothing more than a simple understanding of basic arithmetic and an open mind is needed to understand the key concepts. In essence, a neural network can be thought of as this strange black box into which we enter numbers and (after a series of mathematical processes) receive numbers that help us answer our questions on the other side. The box is trained iteratively until it has learned to give accurate answers for the intended question (e.g is this photo a duck or a lion?). Let us now open this black box...

Nothing but lines, circles and numbers

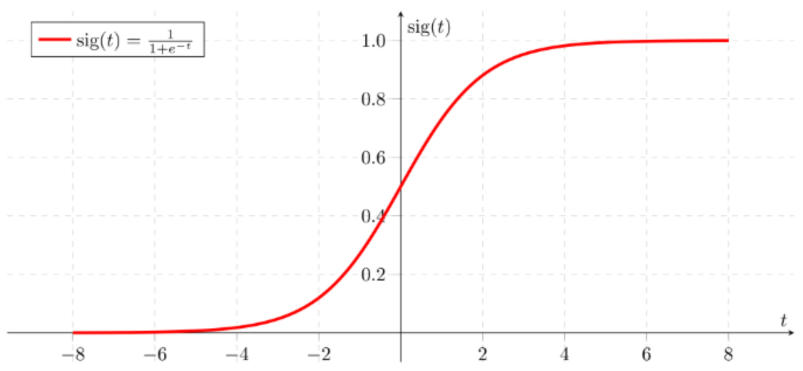
Neural networks act as simpler models of how the brain functions. Inside the brain we find neurons in a complex web that are all connected with one another at synapses. Electrical impulses sent from nerves all around the body are fired through these neurons until they reach some part of the brain that tells us how to react to the stimulus. Of course this is dramatically oversimplified, but it does help give a jumping off point. Neural networks are doing the same thing - they have a series of input signals that are sent down a web of connections that result in a useful output being presented on another side. What is even more wonderful is that this process consists of little more than multiplication and addition.

To better understand this, let us meet the concepts of the weights, biases, and the nodes. My slightly crude image to the right shows how we can represent parts of the network as nothing more than some circles (which we will call nodes) connected by lines (edges). Each of these elements have an associated number. In this scenario we have three numbers at the blue input nodes (0.5, 0.9 and 0.1) These could either be the numbers initially input into the network by the user or results from previous parts of the network. All three are connected to the orange output node by an edge and these too have numbers (2,0.2 and 1). These are the weights, and they determine what to multiply the input numbers by. For example, to traverse from the top input node to the output node here, we multiply 0.5 by 2 to get 1. We do the same with the other two input nodes and add them all together to receive the result 1.28.

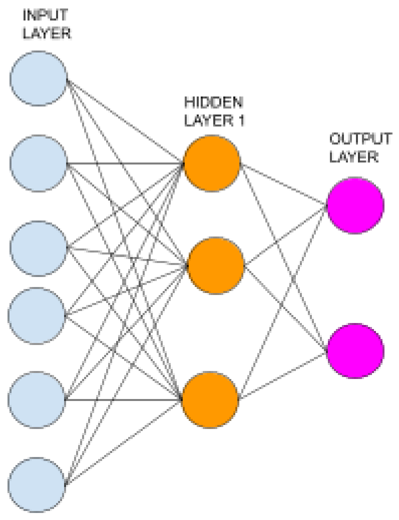


Now there are only a couple more steps to get the final number for the output node. This node also has an associated bias, b, of 1. This bias is then added to the 1.28 giving us 2.28. Finally, we often like to have all our results pushed between 0 and 1. This is very useful in cases where we are determining probabilities relating to how confident a network is (for example how likely it thinks the given input is an image of a lion as opposed to a duck) and prevents certain results from getting too high in a way that makes computation difficult and outputs overly biased results. In order to do this, we use an activation function. These come in many shapes and sizes but here I have used the sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



Note that, for the purposes of understanding, the exact details of the function are irrelevant as long as one notes the mapping between 0 and 1. The sigmoid function also has the added benefit of mapping our numbers in a way that ensures small changes in the weights and biases create small changes in the output (which will come in handy later down the road). So, applying the sigmoid function, $\boldsymbol{\sigma}$(2.28), we get our final result of 0.91!

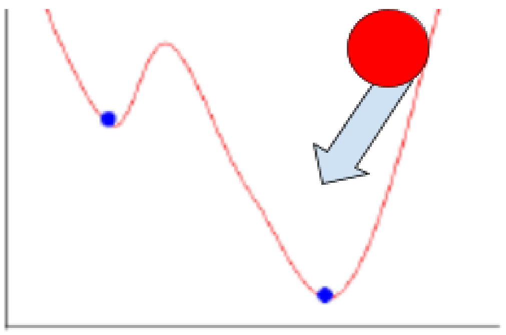


So, let's now expand this. In order to get useful results, we will want to make our network much bigger - as illustrated in the network to the right. It now looks much more complicated but it is still doing the exact same thing as our previous example - just many more times. Our input data is put into the input layer and then we feedforward through the network only by using addition, multiplication and our activation function. Each edge still has its own unique weight and each node still has its own unique bias. If we do this, what we are left with is two numbers between 0 and 1 in our output layer. It is up to the model architect to decide how many nodes to use and this is often found through experimentation. However, the number of input nodes is entirely derived from the amount of input data and the number of output data nodes is decided by the nature of the problem to be solved. A network with two outputs is often found in situations where the network has to decide between two possibilities of results (e.g is the input an image of a lion or a duck?). These outputs can be interpreted as the probability the network believes that that answer is the correct one. For example if the output node that is being used to show the likelihood of the image being a duck shows 0.8, then the network is 80% certain that the image was a duck. The output node with the highest value is chosen as the final result given by the network.

Rolling down the hill

Now we have discovered how a neural network can convert numerical inputs into numerical outputs that can then be interpreted in order to aid a computer system. However, there is a key part of the puzzle missing: if all we are doing is adding and multiplying numbers, how can we receive an accurate and useful answer? This becomes even more puzzling with the information that the weights and biases of the network are often assigned completely randomly! Would this not provide a completely random output? The answer is yes - merely creating the network brings us nowhere closer to building our AI. This brings us to what transforms our useless group of random weights and biases into an extremely useful tool: gradient descent.

This is also how the concept of machine learning comes in. By using gradient descent, our model can learn the optimal weights and biases for the network in order to solve the problem as accurately as possible. This is a form of supervised learning - a type of machine learning in which the model is constantly tested against a dataset and then adjusted based on its performance. Before the training of the model, a labelled dataset will have been compiled for this purpose (for example, a dataset of lion and duck images with the correct answer for each one). We can then pass each example through the network and we compare the network's output with the intended answer to find a 'cost'. For example, imagine we are passing a picture of a lion through our network from earlier. We want the network to ideally give a 1 for a lion and a 0 for a duck. If instead it has 0.4 for a lion and 0.6 for a duck, then there is some cost associated with those values. The closer to the correct values, the lower the cost. Our intention in gradient descent is to minimise cost (meaning a more accurate model).



To understand how this can be done, we can imagine a graph of cost against the value of a weight in the network like the one to the right. In reality, we do not know what this graph actually looks like, but we can work out the gradient (the slope) and this can be used to traverse down the graph to its minimum point. This is best understood by imagining you are a ball on the slope located at the point corresponding to the current weight value and the current cost of the network. Just like a ball on a hill, we can imagine rolling down the hill until its lowest point. At this lowest point, we discover which value for the weight allows that low cost and, having done this to update each parameter in the model, we have achieved an accurate network. Of course, it is extremely computationally expensive to actually simulate a ball rolling down the 'cost curve hill' but the same process can be reproduced with our good friend calculus. From our knowledge of the cost function and the arithmetic used in the network, we can calculate partial derivatives to determine the exact gradient (or steepness) of the cost curve for each parameter in the network. Then, with this gradient we know exactly how much to change each weight and bias in the network in order to lower the cost. For those more mathematically interested, this is possible because the change in the cost is equal to the change in the parameters multiplied by the gradient. So, if we choose the change in parameters to be the gradient times a negative number, then we ensure that the change in cost is always negative (as this will be a negative number times a square number and square numbers are always positive). This process is then repeated again and again until it appears the cost has gotten as low as it can go. Now, we have a finished (and hopefully accurate) network with finely-tuned parameters!

One concept to note is that of local minima. These exist because of the complexity of the cost graph and will mean there are several points where it appears the cost is minimal. In the graph shown above, there are two points marked with blue dots that appear as valleys in the cost curve hill. If our ball started to the left of the graph it would fall into the higher valley to the left and so will not be the best model it can be. To avoid this, when building a model, many should be created with random starting points so that it can be certain that the full space has been explored and the optimal model has been created.

And that's it!

That brings us to the end of our neural network journey. There is still so much more to explore as real models expand on these concepts in order to create all sorts of weird and wonderful AIs and for simplicity I have done away with a lot of the mathematical detail and complexity, but I hope that this paper has provided an easy to access but in depth look into the 'black box' of neural networks. Maths has a really incredible ability to turn simple numbers into beautiful and intricate systems - and for me neural networks are a perfect example of this.

Image references

Sigmoid graph -Everything you need to know about "Activation Functions" in Deep learning models - Vandit Jain

Cost curve - Everything you need to know about Neural Networks and Backpropagation — Machine Learning Easy and Fun - Gavril Ognjanovski