NEURAL NETWORK PACKAGE MODULE 1

WHAT IS A NEURAL NETWORK?

QMIND EDUCATE

ANDREW FARLEY

TABLE OF CONTENTS

YOUR BRAIN

Ever since the dawn of digital computers humans have tried to model intelligent behaviour in machines. There were many approaches formalized in the 1950's, 60's, 70's, and 80's, of which form the basis of Artificial Intelligence today. Some of these techniques include genetic algorithms, regression, and **neural networks**, the latter being what we will focus on in this package.

How it Works

First, let's talk about your brain. Your brain is made up of tiny units called **neurons**. These neurons have **axons and dendrites**. A dendrite receives information for the neuron, while axons transmit information to other neurons. If there is a **strong enough signal** from the dendrites, the **neuron will fire** its axons which are connected to other neuron's dendrites. This causes a chain reaction of many neurons firing one after the other. It is this network of billions of neurons that allow you to think!

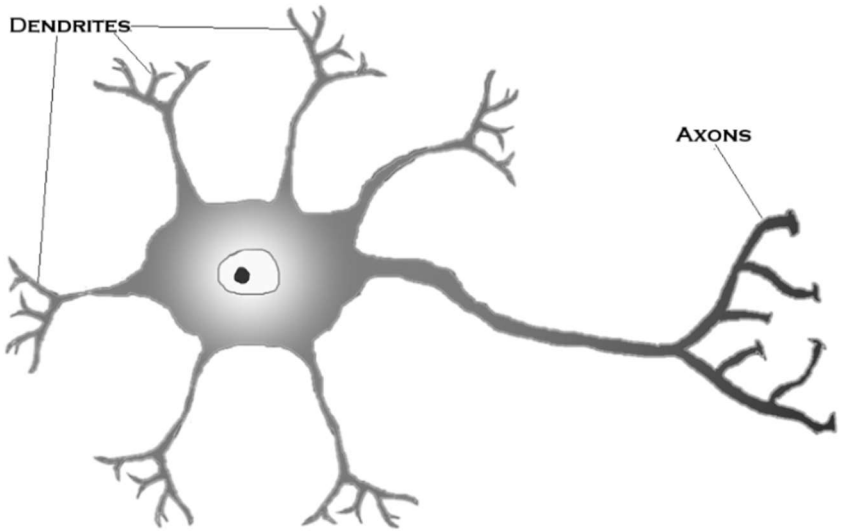


Figure 1: Over-simplified diagram of a biological neuron

How it Learns

Your brain learns by **strengthening different connections** between neurons. When you do something, such as taking a step to walk, different neurons fire and form connections. These connections are strengthened **every time a certain sequence of neurons fire**. When you take a step, a certain sequence of neurons fire and *reinforce* the connections between themselves. You have likely been walking most of your life which is why this process is very familiar to you; the connections between the neurons that make you walk are very strong.

BASICS OF NEURAL NETWORKS

Structure

Enough biology, how could this be translated into something a computer can understand? That is the problem that computational neural networks solve. In a similar fashion to a brain, a basic neural network will have many **layers of neurons**. Each layer will take inputs from the last layer and output to the next layer. Each neuron will take **the outputs** from every neuron on the **last layer** as its inputs and **will output to** every neuron on **the next layer**.

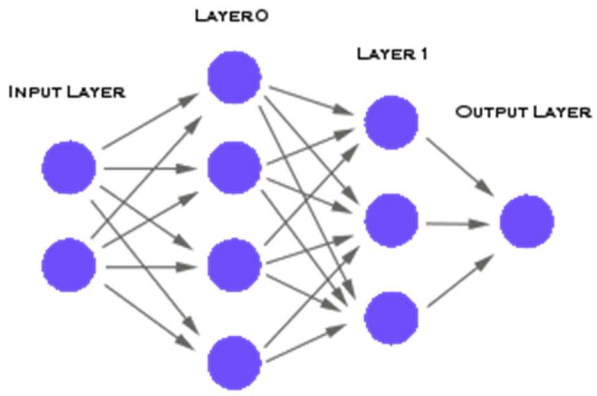


Figure 2: A possible structure of a neural network

How Neural Networks Think

The process of computing an input into an output using these *layers of neurons* is called **Forward Propagation**. When we think of a programmed neuron we imagine it to have some inputs, a numeric weighting for every input, a bias, and an output (which will be fed into the **next layer as inputs**). Unlike the brain, neural networks use floating point numbers and weighting rather than thresholds to pass input on as described above. This allows the network to use many less neurons than a brain as one computer neuron can represent many of the binary biological neurons (just as a decimal number can be represented by many binary digits).

An Aside:

A numeric weighting (as described above) acts as the connection between different neurons. Looking to the example network above, every arrow is really a number between -1 and 1 that will manipulate (through multiplication) the value of the left neuron when feeding it to the right neuron. Any number in this range must be possible (i.e. 0.567) and the way to accomplish this in computers is using something called floating point numbers. This will allow for our continuous value.

INDIVIDUAL NEURONS

Because of the use of *floating-point numbers*, neural networks must operate in a different way compared to their biological counterparts. Rather than only being connected to a few other neurons in the brain, computer neurons output to ***every* neuron** in the next layer. A neuron's output is multiplied by the *weighting* of the connection to the next neuron. The value of these weights is like the connection strength in a biological neuron. The bigger in magnitude it is, the stronger the connection and the larger the influence it will have on the next layer.

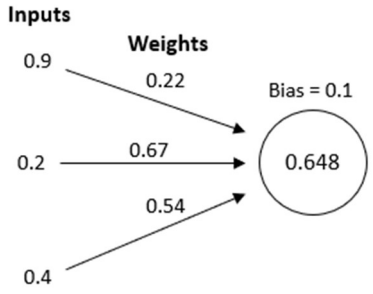


Figure 3: A computer neuron

In the example above, we have a neuron on some layer with weights [0.22, 0.67, 0.54] and a bias of 0.1. In the diagram above, we have the inputs on the left, the weights in the middle, the bias at the top right and the output inside the circle. This whole diagram is a neuron. We can think of every neuron as a small function. This function will take a *weighted sum* of all the inputs, *add the bias*, and output a value as an input to every neuron on the next layer. The output can be seen inside the circle in figure 3. Let's begin by taking the weighted sum. A weighted sum is simply computed by multiplying each input by its respective weight and adding each of these now weighted inputs together to obtain a weighted sum.

$$Weighted\ Sum = 0.9 * 0.22 + 0.2 * 0.67 + \ 0.4 * 0.54 = 0.548$$

Then, we can compute the neuron's output by adding the bias to our weighted sum:

$$Output = Weighted\ Sum + Bias = 0.548 + 0.1 = 0.648$$

NETWORK OF NEURONS

Let's look at an example of these neurons working in tandem to create a neural network:

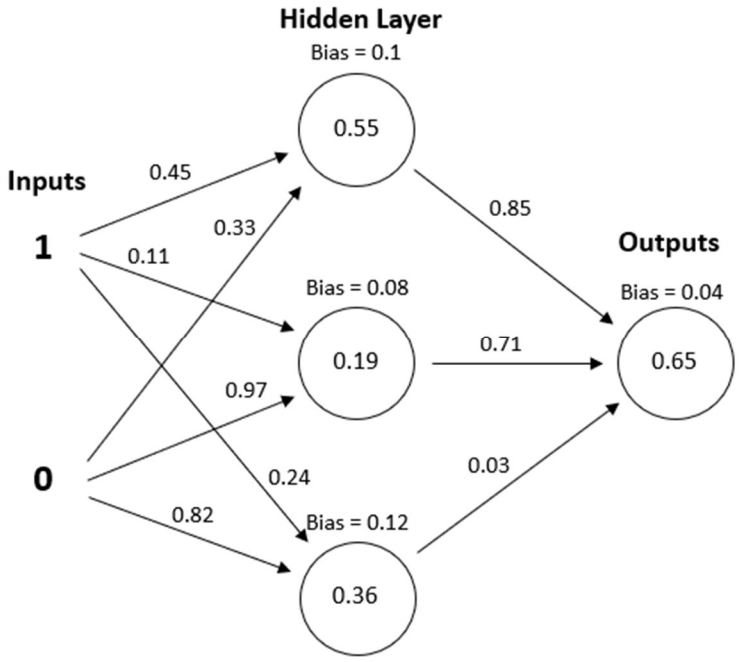


Figure 4: An example of how an input would propagate through a neural network

There is a lot going on in this image, so let's break it down. To begin, this is defined as a **2-layer network**. We define the layer count in a network to be the number of layers with weighting before them. The input layer doesn't count as it does not have any weights connecting to it. We only have one **hidden layer** in the example above, but there could be more. A *hidden layer* is any layer which is not the output or input. As you can see, the input to this network is a matrix [1, 0]. For most neural networks the input will be a one-dimensional matrix which have values ranging from 0 to 1. The idea of a matrix is quite simple, it is generally a list of numbers in a given number of dimensions. When I talk about a one-dimensional matrix, think of it as a line of numbers as a line has one dimension. Each input is fed through to every neuron in the next layer and multiplied by the weight between the current neuron and the one in the next layer, as previously discussed. Each neuron in the next layer will then **sum these products and add the bias** to produce its own output. Looking at the top neuron in the hidden layer:

$$Output = Weighted\ Sum + Bias$$

$$Output = Input\ 1 * Weight\ 1 + Input\ 2 * Weight\ 2 + Bias$$

$$Output = 1 * 0.45 + 0 * 0.33 + 0.1$$

$$Output = 0.55$$

The output for every neuron in the layer is then computed. These outputs are then passed to the next layer, in this case the output layer. The same process happens as before; each output from the previous neurons are multiplied by the weight of the current neuron and summed. This is what is meant by the outputs of the previous layer are the inputs to the next. Once we have the weighted sum, we add the bias which provides our final result!

$$Output = Weighted\ Sum + Bias$$

$$Output = Input\ 1 * Weight\ 2 + Input\ 2 * Weight\ 2 + Input\ 3 * Weight\ 3 + Bias$$

$$Output = 0.55 * 0.85 + 0.19 * 0.71 + 0.36 * 0.03 + 0.04$$

$$Output = 0.6532$$

NEURAL NETWORKS AS FUNCTIONS

This entire network of neurons can be thought of as **one unique function** with many inputs. The exact number of inputs to this function is the total number of weights plus the total number of biases. Both these values can be manipulated to change how the network behaves which is why we consider them part of the function. In the case of figure 4 the network would have 13 inputs if it were thought of as one function, 9 weights plus 4 biases. We will see why this is so important later. The problem is, the current "function" (or neural network) we have been describing has one major flaw; it is linear! This means it won't be able to adapt very well to many real-world examples we will be training it on as most things in life do not have a linear relationship. For example, this network would do a very poor job of predicting where a ball is when thrown. The ball's trajectory will be parabolic, and a line is not able to fit this curve well. To fix this, we add something called an **activation function** to each neuron. This function will take what the neuron outputs and apply some sort of non-linear function to the value. We will discuss why this is important and useful in the paragraph below. In Figure (5), you can see what our computer neurons will look like with this function included. This is the exact same neuron as highlighted in the Figure (3) example above, but now with an activation function included.

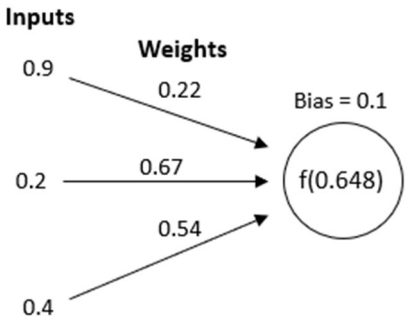
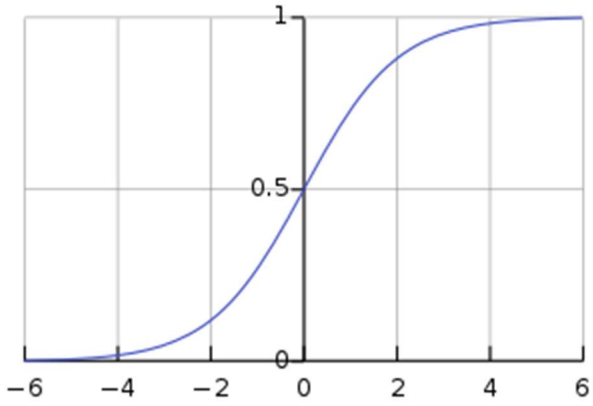


Figure 5: Computer neuron with activation function

Not much is different except that our output of 0.648 is now being passed into some activation function *f*. The result of *f* is what the *neuron will now output*. One of the first *activation function*s used with neural networks is called the **sigmoid function.**

$$Sigmoid\ Function: f(x) = \frac{1}{1 + e^{-x}}$$



We use the sigmoid function for multiple reasons. The first is that it has a smooth and simple derivative which we discuss later. The second is that it will map any input to a value between 0 and 1. It will give us a probability! This is so much more useful than a step function (a function with a discrete output of 0 or 1) as we can have neurons that are 20% or 50% activated rather than just turned on or off. As you can see, when *x* is equal to *zero*, we get an output value of *one half*, or 0.5. This function, like most others, assumes and somewhat requires all weightings to be in the range -1 to 1 (i.e. they can be negative). This function is clearly not linear which can lead to smoother and more realistic predictions in the overall output of the network because it provides a percentage of activation. The sigmoid function is somewhat outdated and many newer

activation functions have been tested to be more successful, these newer methods will be touched on in the next module.

How Neural Networks Learn

A neural network learns in a similar way that your brain does, by *strengthening connections*. There are many ways that the network can learn (generally referred to as **machine learning**), but the one we will focus on is called **gradient descent**. This process is a form of **supervised learning** where the machine *adjusts the weightings* in each layer of neurons based on the *error* of its current output and weightings.

In a process called **backpropagation**, every layer is transformed based on how each node would have contributed to the overall error of the final output. This is where the tweaking happens. Every time backpropagation is run we first pass data in (as inputs) of which we know the desired output, often referred to as **labelled data**. We then calculate the error between what the network is currently outputting and what the desired output is. We then tweak the *weightings* and *biases* very slightly so they will produce a *more accurate result* on every layer. It often takes many iterations of backpropagation to "train" a neural network well enough to be accurate, therefore we need large amounts of *labelled data*. The weights converging to an optimal value is analogous to us strengthening the connections in our brains; both processes won't always yield perfect results, but the more practice and testing done, the better the performance.

MODELLING THE 'AND' OPERATOR

To see this in action, let's assume we want to make a neural network to model the binary AND operator. The AND operator takes **two binary inputs** (a zero or a one) and will produce **one binary output**. An AND operator requires that both binary inputs be a 1 (or true) to output a 1. You can think of these 0's and 1's as neurons in your brain firing or not: if a 1, the neuron will fire, if a 0, it will not. By this definition, for an AND operator, both input neurons must be firing for the output result to be firing. If either or neither of the input neurons are firing, the output will not fire. This behaviour may be demonstrated in what is called a **truth table**. The *truth table* for this behaviour may be seen below.

Input 0	*Input 1*	*Output*
0	0	*0*
0	1	*0*
1	0	*0*
1	1	*1*

Figure 6: Behaviour of the binary 'and' operator

If any zero is inputted the output will be a zero. Only in the case where the inputs are both one will this function output one. To model this, we first create a neural network with randomly

assigned weights and biases. For now, let's assume we're creating a network that has 2 inputs, 1 hidden layer of 3 neurons, and 2 output neurons. The reason there are 2 output neurons instead of one is the AND operator has **2 possible outputs**, 0 or 1. A neural network always provides a **list of probabilities** of how likely the input matches to a certain output. We can then find the highest probability and say that is what the network thinks the result should be. Here is our randomly initialized neural network with a sample input and output:

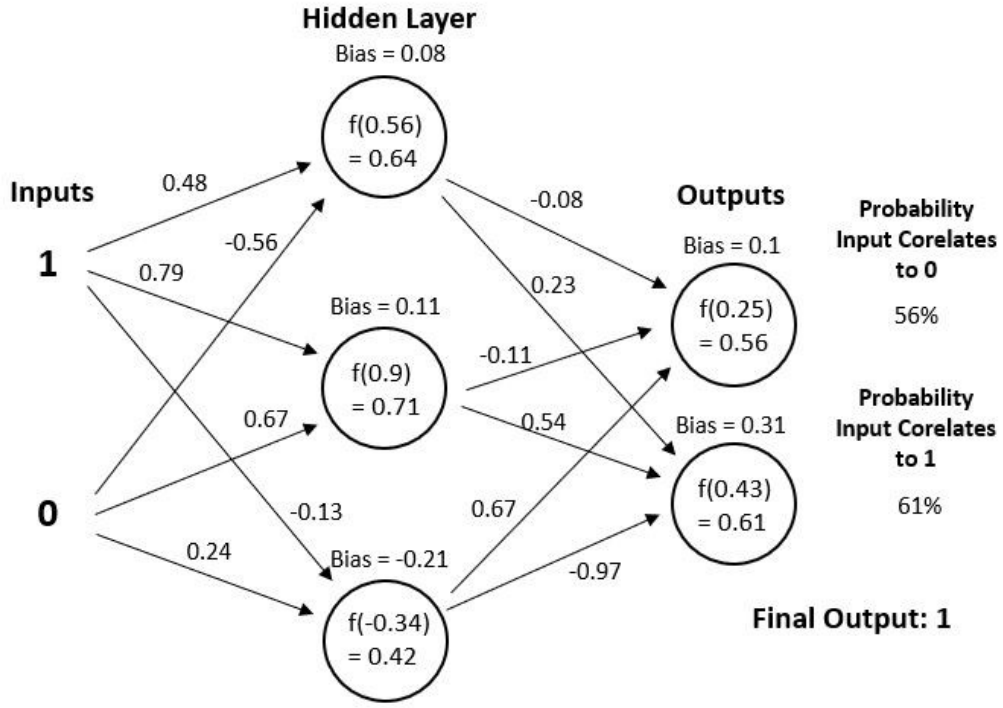


Figure 7: Neural network trying to model the 'AND' operator

Every weight and bias have been randomly initialized in the range -1 to 1 inclusive. As we can see, the input matrix [1, 0] has produced a probability of output 0 of 56% and a probability of output 1 of 61%. Therefore, this network thinks that the input of [1, 0] should result in a 1 because it has the highest probability. This is incorrect for the 'AND' operator! We need to train the network so it will give us the output we desire.

As a small side note: the output probabilities are independent of one another (i.e. the computation of one won't determine the result of another). We can change that by using a different activation function.

By looking at this network, what connections (or weights) do we need strengthen or weaken to get the output we want? This can be very confusing; many questions are probably popping into your head right now. Let's start by looking at the output layer. We know we need to increase the

weights going into the top node and decrease the weights going into the bottom node. For now, we can take some constant number (n = 0.05) to add to the weights going into the top node and subtract from the weights going into the bottom node. By doing this, we have rewired the network and therefore changed the output:

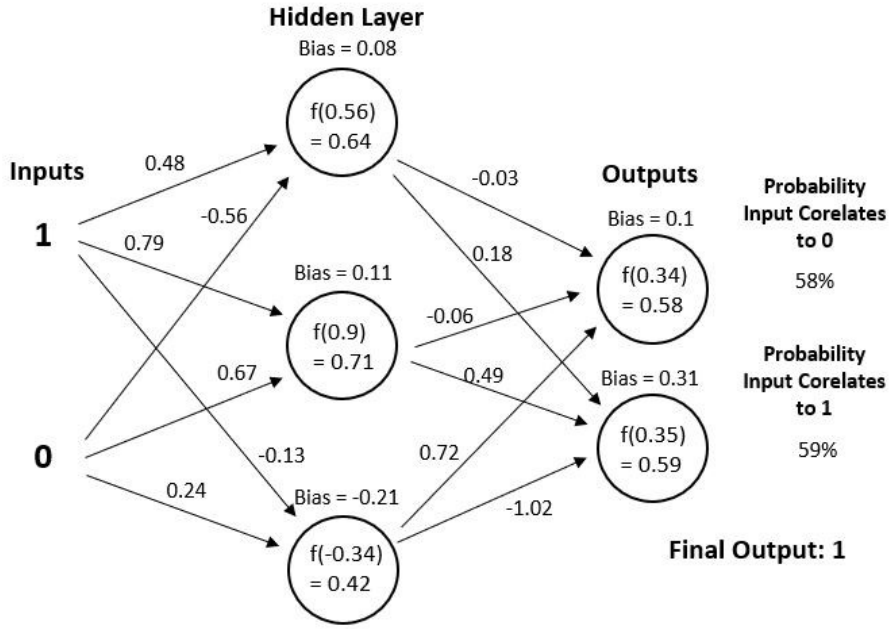


Figure 8: Neural network after weighting tweaks

As you can see, this changed the output probabilities a lot! It is still producing the wrong result, but we moved a step in the right direction. What if instead of a constant value, we based the change in weighting off the **error** from the expected result and our actual result. In this case, the expected output matrix is [1, 0]. Our network outputted [0.58, 0.59] so the error would just be the expected minus the actual.

$$Error = [1, 0] - [0.58, 0.59] = [0.42, -0.59]$$

ADJUSTING WEIGHTS IN A LINEAR NETWORK

We also know that the current weights represent the influence that one connection has on the output. Therefore they also have an **influence on the error** based on their magnitude and direction. We can use what the layer currently outputs (from weighted sum and bias) and the error to determine how much the weighting needs to change. In a **linear network** (i.e. no activation function):

$$Delta\ Weight = Current\ Output * Error$$

But what about the hidden layer? How will we change those weights? We can get the error for the previous layer (the hidden layer in this case) by propagating the current error backwards with the **new weights**. In a formula:

$$Next\ Error = Error * (Current\ Weight + Delta\ Weight)$$

Then we repeat the process for every hidden layer, saving the weight changes as we go along! This is the process when *no activation function* is used. But what if an activation function is used? How do errors propagate then?

Propagating Error With Activation Functions

Unfortunately, propagating error with activation functions is a little more challenging. In this case, we will also need to find the error **the activation function will contribute** to the overall error, much like we did with the weights. To do this we are going to use an old friend from calculus, the derivative. By definition the derivative is the **rate of change** of a function. If we use what the network outputs at a given layer, we can just plug that into the derivative of the activation function. This will give us the error the function contributes because the magnitude of the rate of change is proportional to how much error the function will introduce. Think about it this way: if we move a small step when the rate of change is large then the output will change a lot. If we move a small step when the rate of change is low, then the output won't change that much. The amount the output changes is where our error comes from! We can just multiply this derivative by the error we're multiplying the weights by to get our resulting error. Let's look at the equation this will yield to clarify.

$$let\ f'(x) = derivative\ of\ the\ activation\ function$$

$$Delta\ Weight = Current\ Output * Error * f'(Previous\ Layer's\ Output)$$

This formula stays constant no matter what layer we are on. What we are doing when we multiply the error and the derivative of the activation function together is producing something called the **gradient**. This is a mathematical term for a vector containing the rate of change of each dimension in a function and will be discussed more in depth in the backpropagation modules as well as the mathematics module. The reason the total error at every layer is the gradient is because when we take the derivative of the activation function we are using every value. For now, just know the process of backpropagation involves going through every layer (starting at the last one and working your way to the front) and modifying the weightings between neurons

based on error. Connections between neurons will be strengthened and weakened, just like in your own brain.

Neural networks are an incredibly cool concept, if I haven't already convinced you so. The fact that we can imitate our own brains using computations continues to astound me. What is even more surprising is that a neural network isn't magical, it is simply a function. We use calculus to optimize the network in a way very similar to how we would anything in a first-year calculus class.

OTHER RESOURCES

If you like learning from videos, I would highly recommend 3blue1brown's series on deep learning (found here: https://www.youtube.com/watch?v=aircAruvnKk&t=1016s) and Daniel Shiffman's series on neural networks (found here: https://www.youtube.com/playlist?list=PLRqwX-V7Uu6Y7MdSCaIfsxc561QI0U0Tb). Most of my knowledge in this field comes from these two sources. They are amazing and I suggest you check them out.

REFERENCES

[1] D. Shiffman, "Introduction to Neural Networks (Video Series)," Coding Train, 2017.

[2] G. Sanderson, "Deep Learning (Video Series)," 3blue1brown, 2017.

[3] A. S. Walia, "Activation functions and it's types-Which is better?," Towards Data Science, 2017.