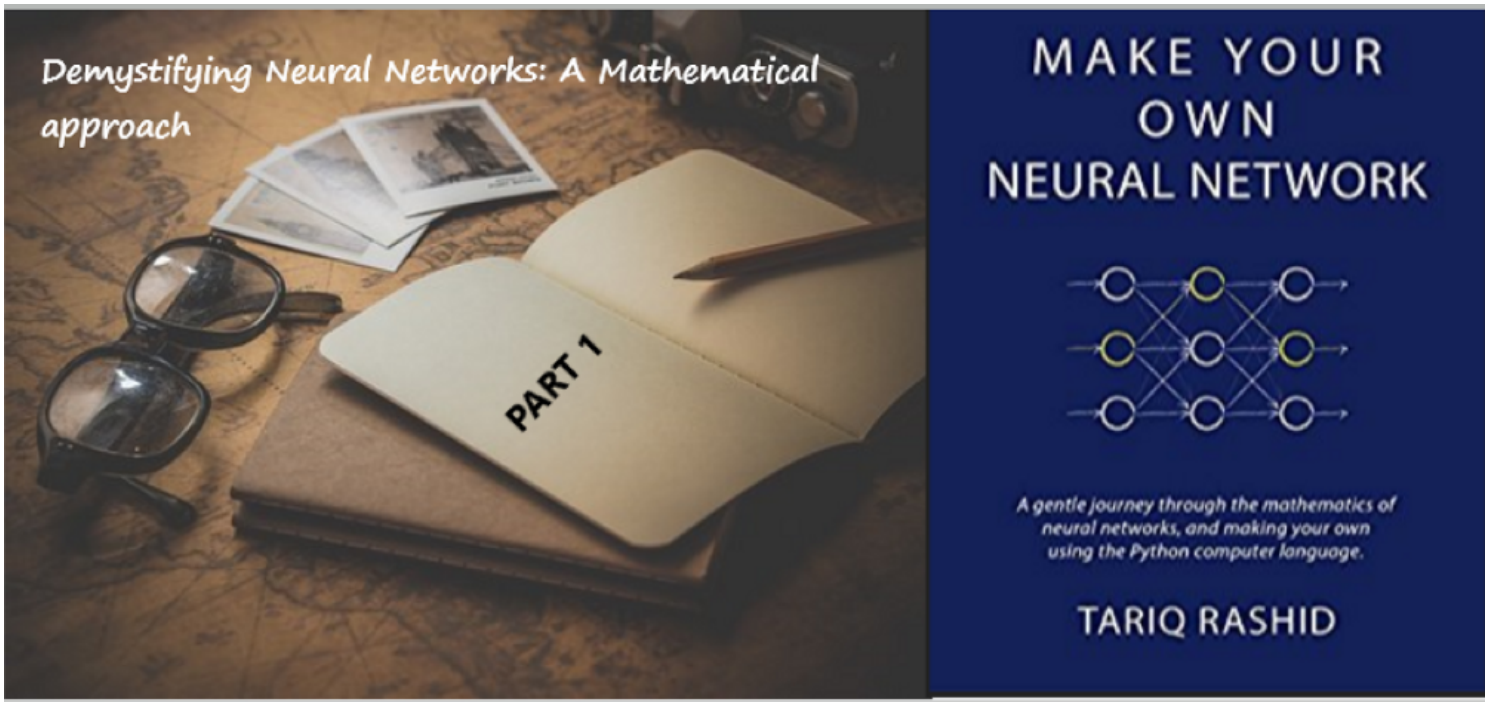
Parul Pandey Follow
Data Science Enthusiast | Writer | Electrical Engineer Really passionate about using Machine Learning for solving real-life problems.
Oct 9 · 16 min read

Demystifying Neural Networks: A Mathematical Approach (Part 1)

*My notes from the book '**Make your own Neural Network**' by **Tariq Rashid.***



Sources: Pixabay/Dariusz Jankowski, link

> *"Take inspiration from all the small things around you."- Tariq Rashid*

There are some tasks which are very simple for computers but hard for humans, like multiplying millions and trillions of numbers, i.e. repetitive tasks. On the other hand, humans outperform computers fairly easy at tasks like recognising faces in a photo. It is at these tasks where we want

computers to get better. Solving problems like this forms the basis of Artificial Intelligence.

Tariq Rashid in his book, **Make Your Own Neural Networks**, aims, to present neural networks in their simplest form. He believes neural networks are always presented as a black box without anyone really getting into the details of explaining how they work. The book emphasizes that neural networks are nothing but simple mathematical manifestations driven by algebra and maths.

I have personally found his book very useful, so I decided to pen down my thoughts in the form of this article for you. After getting a grasp on the fundamentals of a neural network, one can easily apply these ideas to multiple other problems.

. . .

In this article, we will cover the mathematical concepts behind simple neural networks. There won't be any programming to keep things simple. **The main purpose is to show how school level mathematics can be tremendously powerful in making our own artificial intelligence models** which mimic the learning ability of human brains.

Before diving straight into the neural network, we need to look at its basics. We will start with a simple predictor and a classifier which form the foundations of Neural Networks, and then move on to the neural network itself.

. . .

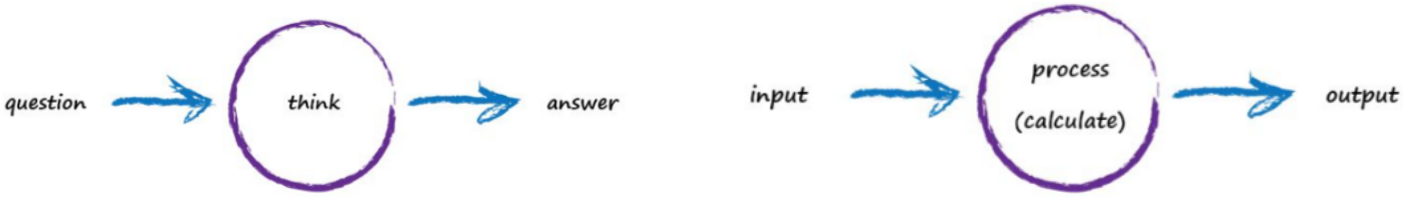
The flow of the article:

- Simplifying a Predictor
- Simplifying a Classifier
- Training a Classifier
- Simplifying a Multi-Classifier
- Introduction to the Neural Network Architecture

. . .

Simplifying a Predictor

If we were to visually represent the problem-solving process adopted by both humans and machine, it would be something like this:



Left: How humans solve a problem, **Right**: How a computer solves a problem

Let us understand how a machine would possibly solve a problem with an example.

Problem Statement:

Consider a machine that converts **kilometres into miles**. It could be a calculator or any simple device**.** Representing the statement diagrammatically, we get :

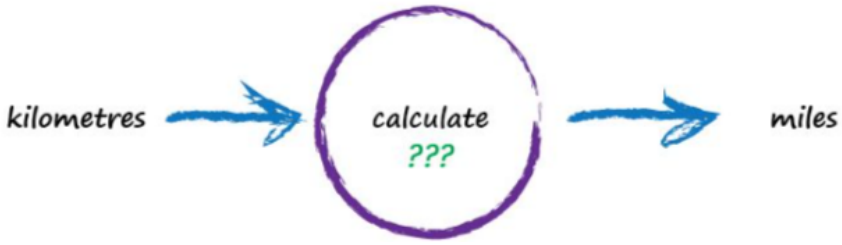


Figure 1

What we know:

From our general knowledge, we know that **doubling the distance in miles, also doubles the distance in kilometres**. Thus, if we were to fit a mathematical equation between miles and kilometres, it would be something like this:

`miles = kilometres x c` , *where* ***c*** *is an unknown constant.*

From real world observations we can create a `truth table,` as follows:

Truth Example	Kilometres	Miles
1	0	0
2	100	62.137

truth table/training data

Our Aim:

Our main aim is to calculate the. `constant 'c'` This is because once we know the value of c, we can plug in that value in the equation: `miles = kilometres x c` , and convert any value of miles into kilometres and vice versa.

Steps

We will start with some random value for **'c'** and calculate the error value for the second truth example. Let us consider three cases of constants with values: `0.5, 0.6 and 0.7` respectively. Calculations have been tabulated below for better understanding.

SCENARIOS	kilometers	Value of constant 'c'	miles(calculated:kilometers X c)	miles(actual)	Error(actual - calculated)
CASE 1(c = 0.5)	100	0.5	50	62.137	12.137
CASE 2(c = 0.6)	100	0.6	60	62.137	2.137
CASE 3(c = 0.7)	100	0.7	70	62.137	-7.863

- **Case1**: Choosing **'c'** as 0.5 gives an error value of around 12 units and the calculated value is quite less as compared to the actual value. To decrease the error, the calculated term should be of higher value. We also know that miles and kilometres are **linearly** related, so **increasing 'c' will increase the output**.
- **Case 2**: Increasing **'c'** to 0.6 decreases the error. This tempts us to increase the value of **'c'** further to 0.7.
- **Case 3**: On increasing the value of 'c' further, we tend to **overshoot** the correct answer. Not desirable again. Hence, 0.6 was a better choice than 0.7. What if were to increment 'c' in Case 2, by just a very small amount like 0.1. Let's see our results.

SCENARIOS	kilometers	Value of constant 'c'	miles(calculated: kilometers X c)	miles(actual)	Error(actual - calculated)
CASE 4(c = 0.61)	100	0.61	61	62.137	1.137

That's much much better than before. We have an output value of 61 which is only wrong by 1.137 from the correct 62.137.

Figure 2

Learnings

This is the way a **simple prediction algorithm** works with a given truth table. The truth table is commonly known as the **training data**. The algorithm estimates the value of constants and iterates over and over again by adjusting the value of **parameters(**in this case **'c')** based on how wrong the model is compared to known true examples. We have simplified a Predictor here.

. . .

Simplifying a Classifier

Let us now try to simplify a classifier the way we simplified a prediction algorithm above.

Problem:

Consider the graph below which shows the measured widths and lengths of garden bugs.

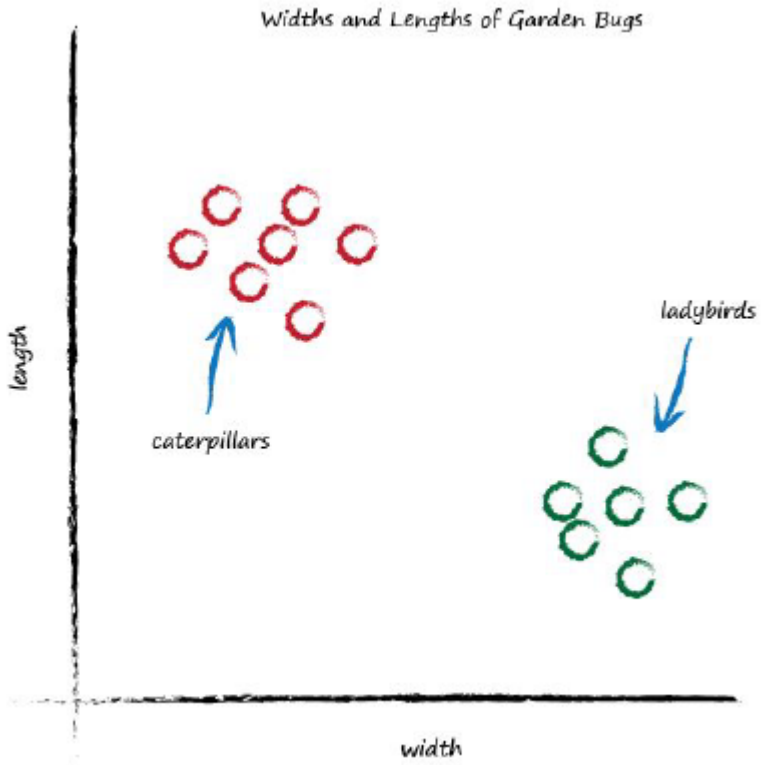


Figure 3

*We observe two categories of bugs. The caterpillars are long and thin, while the ladybirds are wide and short. If a new bug enters the scenario, how can we possibly classify it into a caterpillar or ladybird(**given these are the only two categories available**)?*

What we know:

We know linear functions produce straight lines when their output is plotted against their input. In the case of the predictor discussed in the previous example, **'c'** denoted the adjustable parameter which defines the slope of that line.

Our Aim:

We want to find a line that can correctly classify any unknown bug that enters the garden into caterpillars or ladybirds.

Steps:

There can be three scenarios if we randomly placed a line over the given graph.

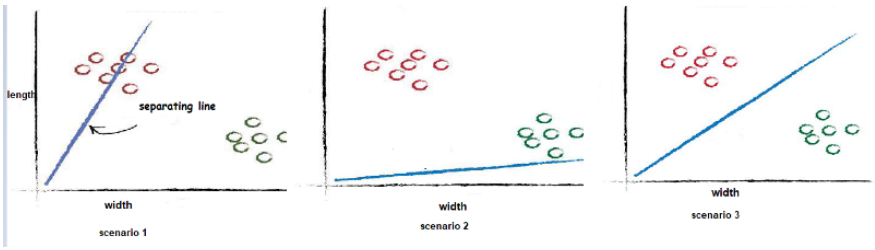


Figure 4

Scenario 1 and **Scenario 2 do not do a good job** whereas **Scenario 3** successfully separates the two categories of bugs. Now if a new bug were to be placed in the garden, it could be easily classified as either of the two known categories depending upon its position above or below the line.

We've seen how a linear function (a straight line in this case**) can be used to classify previously unseen data. But how do we decide where to place the line, i.e. how do we determine the slope of the line. The answer to that will be explained in the next section.**

. . .

Training a Simple classifier

Why do we want to train our classifier? It is because we want our classifier to learn to correctly identify the bugs as caterpillars or ladybugs. And how does training help? Well, we saw in the case of the predictor (from the previous section), that the model was trained on the **real world examples/truth table** aka **Training data** and the error obtained provided the feedback. We shall follow the same principle in the case of a classifier too.

Training Data

From the plots in, `figure 4` it is quite evident that the various dividing lines in all the three scenarios can be obtained by merely **adjusting the slope** of a single line. For training purposes we would require data which is tabulated and visualised as follows:

Example	Width	Length	Bug
1	3.0	1.0	ladybird
2	1.0	3.0	caterpillar

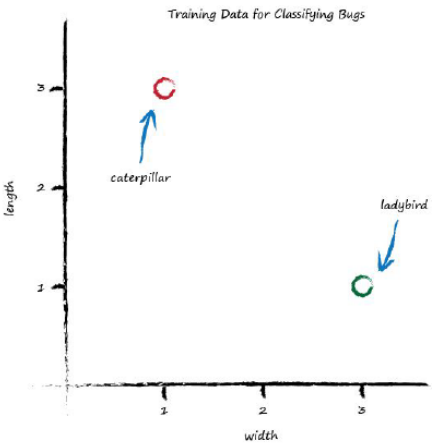


Figure 5 |Left: Training Data/Truth Table |Right: Plot

The training data consists of two examples with their width and length given in centimetres.

Process

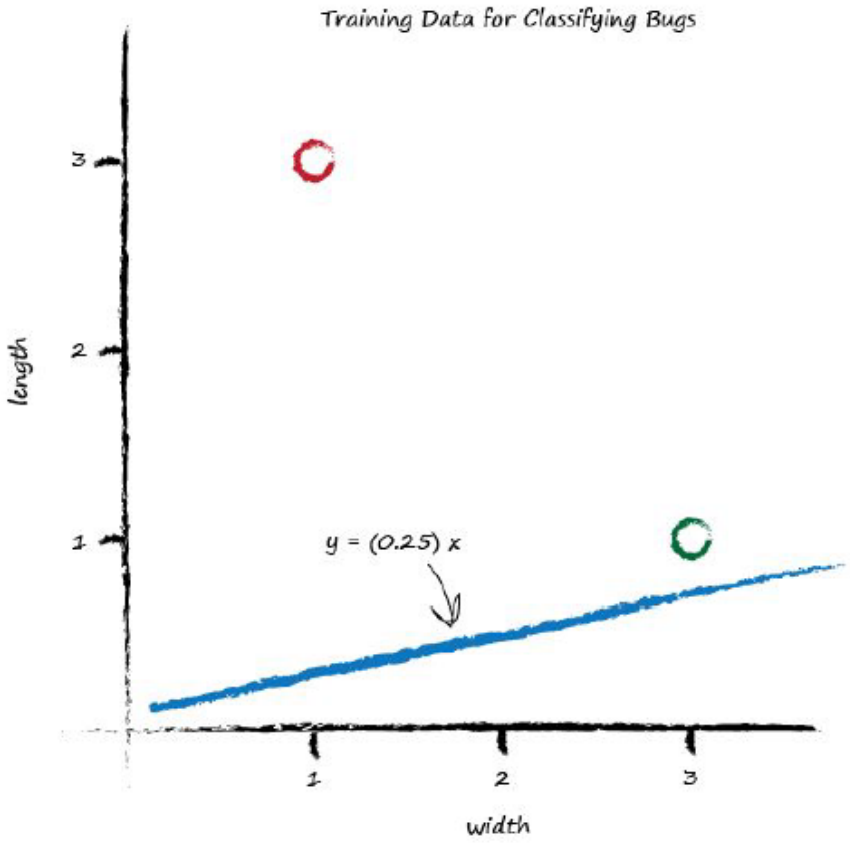
- Since we aim to find a line that can correctly classify any unknown bug into caterpillars or ladybirds; we can start by plotting a random line. The line can be represented by the equation:

$$Y = Ax$$

An equation of a line passing through the origin,

y and x denote the length and width of bugs respectively, and A represents the slope of the line.

Let us assume **A** to be **0.25**(random guess), so the equation of the line becomes **Y = 0.25x.** Plotting the line:



plot for y= 0.25x

This isn't a good classifier as it is unable to classify the bugs. The next obvious step for us would be to increase the slope of the line, but instead of choosing slopes randomly we will **devise a way to update the slope/parameter A** so that our model can learn from the errors.

- **Learning with the first example**

Example	Width	Length	Bug
1	3.0	1.0	ladybird

If we tested the **y = Ax** function with this example where x is 3.0, we'd get

$$y_{calculated} = (0.25) * (3.0) = 0.75$$

```
NOTE:

1) Here y corresponds to length of the bug while x is the
width.

2)Here A is set to the random value of 0.25 as discussed
above

3) x = 3.0, from the training data of first example
```

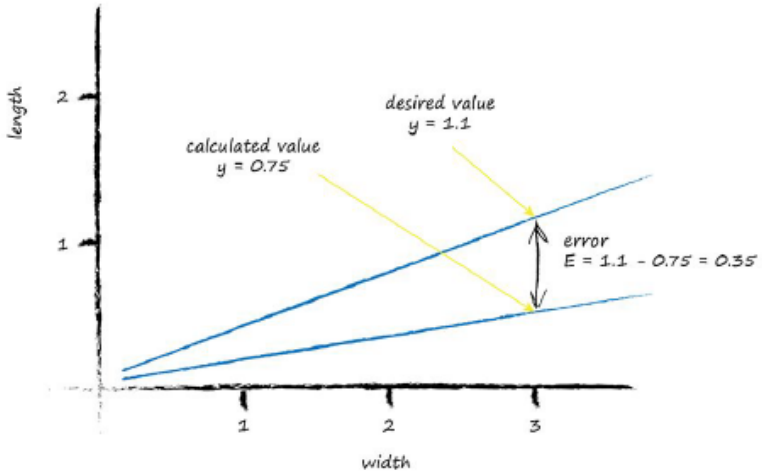
The calculated value of y is much less than the actual value which is 1. Let's think what should be the value of y if not 0.75.

- If y were equal to 1, it would mean the classifier line would pass through the point where the ladybird sits at (x,y) = (3.0, 1.0). We don't want that. We want the line to go above that point. Why? Because we want all the ladybird points to be below the line, not on it since the line needs to be a divider line.
- So let us nudge a little above 1 and choose y = 1.1, when x = 3.0 and calculate error.

```
error = (desired target - actual output)

E = 1.1 - 0.75 = 0.35
```

A visual representation will help us better understand these facts:



- **Error**

The important and an obvious question here is that how is this **error** value helping us to refine the slope of the line/parameter A. It will become clear in the following section.

We know, the linear function of a classifier is given by the equation:

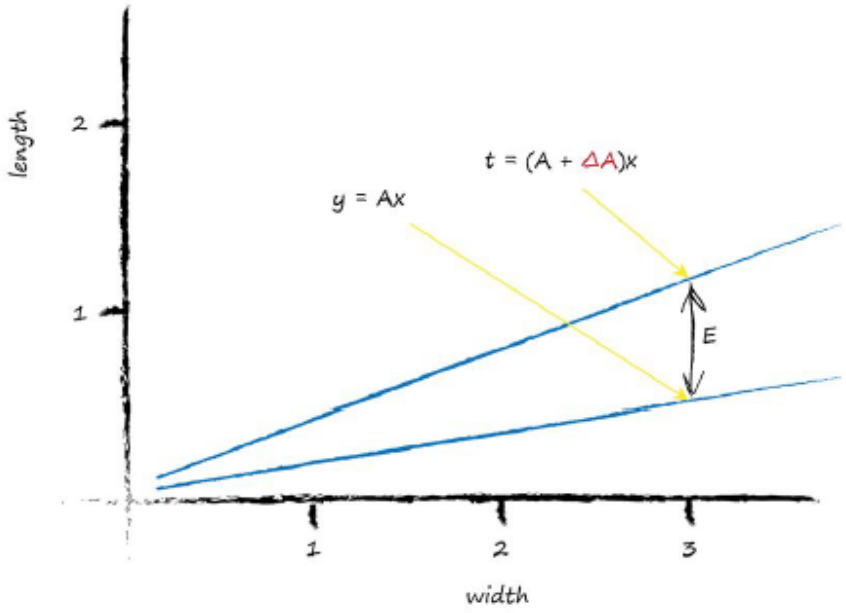
$$y = Ax$$

Let's call the correct output value to be 't'(t stands for target value). To get that value t, we need to adjust A by a small amount.

$$t = (A + \delta A)x$$

New slope is given by:

$$(A + \delta A)$$



Remember the error E was the difference between the the correct value and the one we calculated based on our current guess for A. That is,

$$E = t - y$$

Combining all the equations and solving for the parameter A:

$$E = t - y = Ax + (\delta A)x - Ax$$

$$E = (\delta A)x$$

$$\delta A = \frac{E}{x}$$

That is it. We have found the relationship between **the error** term and the parameter **A**. We can use the error E to refine the slope A of the classifying line by an amount δA. This is precisely what we wanted to know, i.e. how much to adjust A by to improve the slope of the line, so it is a better classifier, being informed by the error E.

Going back to the example and plugging in the values in the formula, we get:

$$Error = 0.35, x = 3.0$$

$$\delta A = \frac{E}{x} = \frac{0.35}{3.0} = 0.1167$$

This means we need to change the current A = 0.25 by 0.1167 and the new improved value for A is (A + δ A) which is 0.25 + 0.1167 = 0.3667. As it happens, the calculated value of y with this new A is 1.1 and as you'd expect it's the desired target value.

Finally, with all these calculations, we have a method for refining the parameter A, informed by the current error and thereby helping us to decide the slope of the line.

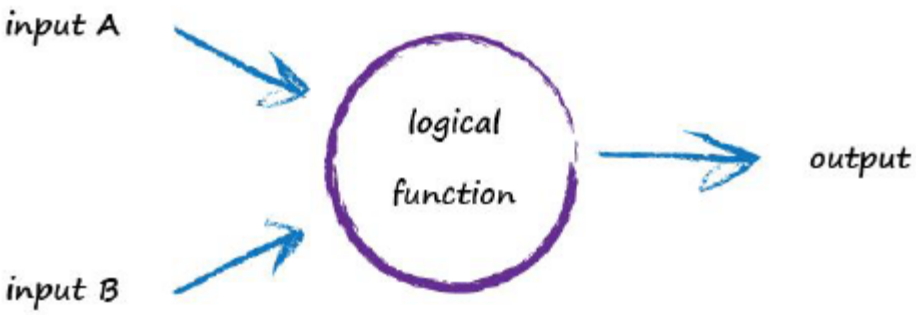
. . .

Simplifying a Multiple Classifier

A neural network is made of many classifiers working together. Even though a simple classifier is quite useful, it does not provide a solution in all the cases.

Limits of a Simple Classifier

We shall study limits of simple classifiers with the help of **Boolean functions**. Boolean logical functions typically take in two inputs and produce a single output. A typical Boolean logic function can be represented as:



Representation of a standard Logic Function

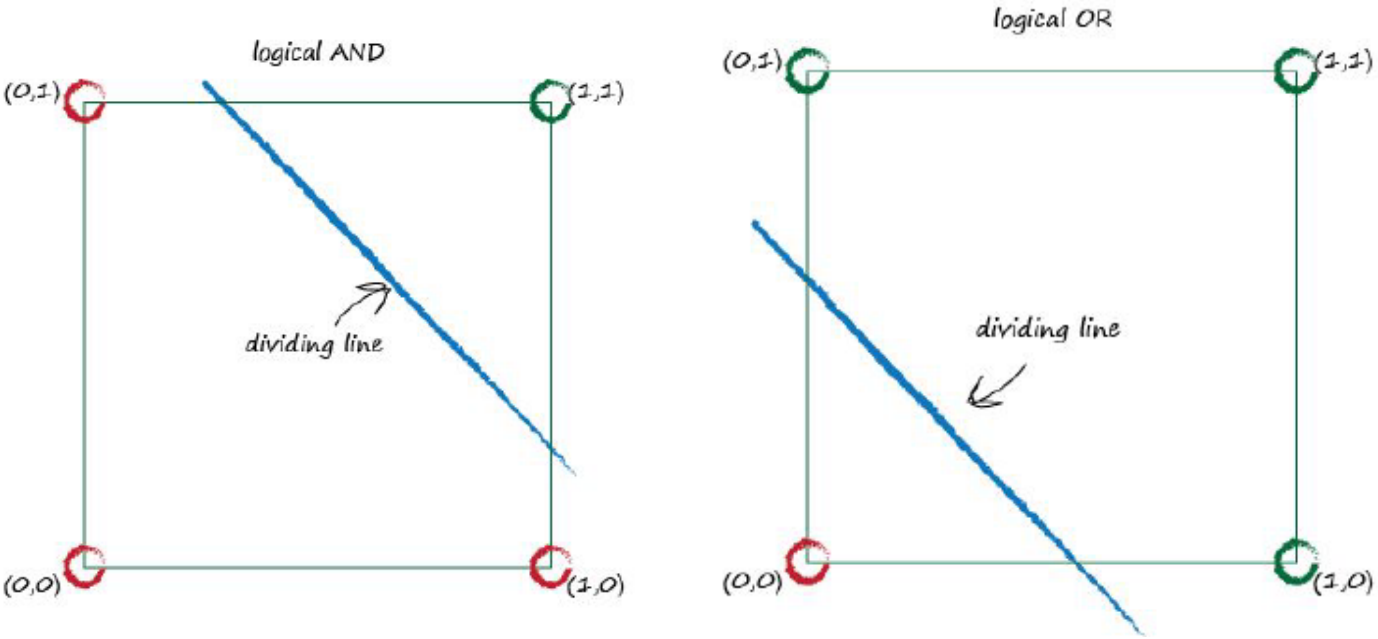
Logical AND and OR

We can represent the **logical AND** and **OR** functions in the form of a Truth Table with inputs A and B.

Input A	Input B	Logical AND	Logical OR
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

- The **AND** function is only true if both A and B are true
- The **OR** is true whenever any of the inputs A or B is true

The Logical functions can also be represented in the form of a graph with the two inputs A and B as coordinates on a graph. True outputs are shown in green while False are shown in red.



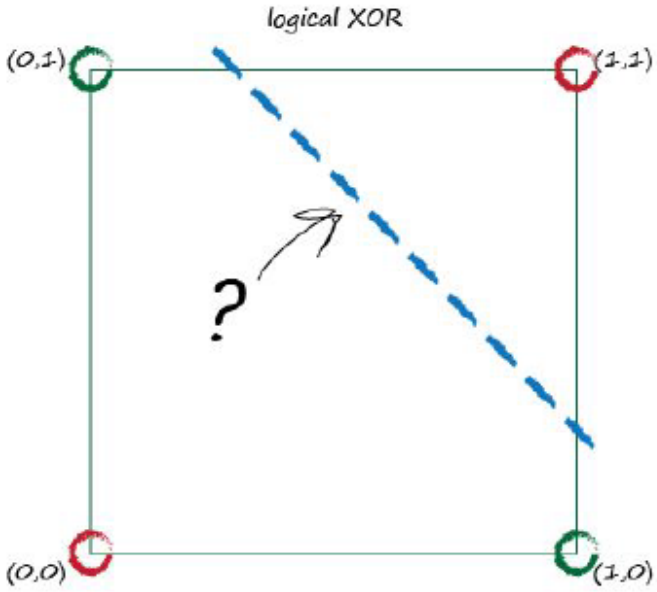
We can even draw a straight line which acts as a linear classifier seamlessly dividing red area from the green one. There are many variations of this dividing line that would work just as well, but the main point is that it is indeed possible for a simple linear classifier of the form **y = ax + b** to learn the Boolean **AND/OR** function.

XOR gate

XOR stands for **eXclusive OR**, which only has a true output if either one of the inputs A or B is true, but not both.

Input A	Input B	Logical XOR
0	0	0
0	1	1
1	0	1
1	1	0

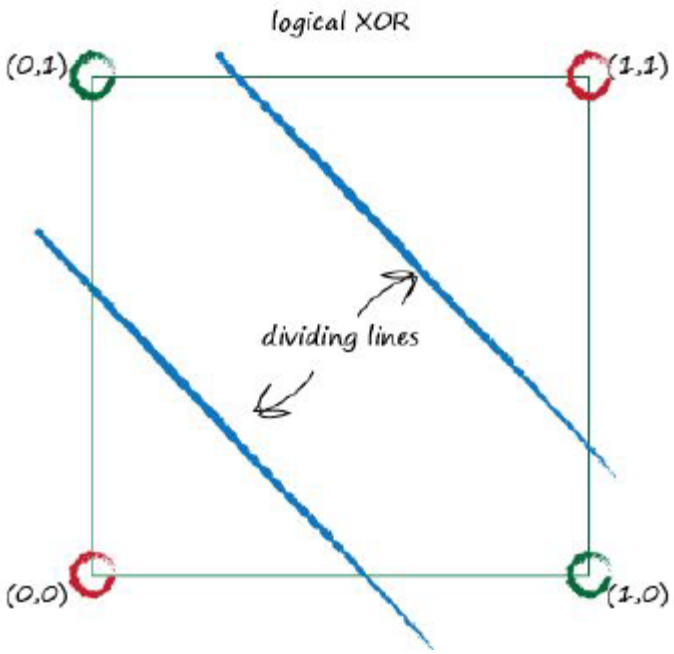
Drawing the same plot as above with inputs as coordinates, we obtain a plot as below:



There seems to be a problem since we are not able to separate the green region from the red one with a single linear line. In fact, there appears to be no single straight line that can achieve this task. **Thus, a simple linear classifier can't learn the Boolean XOR if presented with training data that was governed by the XOR function**. This appears to be a significant drawback of linear classifiers, and they become redundant if the underlying problem is not separable by a straight line.

What's the Fix?

Not all problems are linear. So how do we solve such problems? If in the above plot, instead of a single straight line, we use two straight lines, it seems to fulfil our purpose. This is like multiple classifiers working together, and this also forms the **underlying principle on which neural networks** are based.

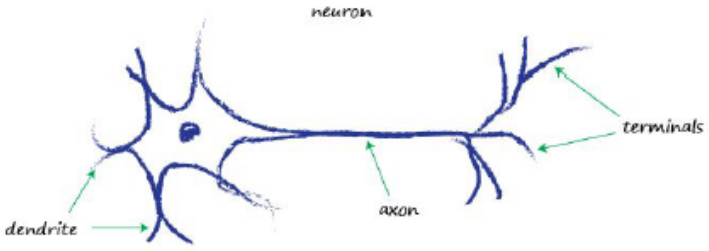


. . .

Simplifying a Neural Network

Neural networks are nothing but many classifiers working together. But, before discussing that, let us learn a bit or two about animal brains which inspired the neural network approach.

Neurons—The building blocks of a neural network



a biological neuron

It has been found that neurons transmit electrical signals from one end to another, through dendrites along the axons to terminals. The signal travels through various neurons, and this is how our body senses light,

sound, pain etc. Signals are transferred from sensory neurons along the nervous system to our brains for it to elicit the desired response.

Working of a neuron

A neuron takes in an electrical input and transmits another electrical input mimicking exactly how a classifier/predictor works. However, neurons cannot be represented as linear functions. This is simply because a neuron doesn't produce an output that is a linear function of the input unlike simple classifiers or predictors which follow the equation: ***output = (constant * input) + (another constant).***

Activation Functions

It has been found that neurons do not act instantly but only once they reach a certain. `Threshold` This is primarily because the neurons do not want the noisy and tiny signals to pass through. A function that takes the input signal and generates an output signal, but takes into account some kind of threshold is called an. `Activation function`

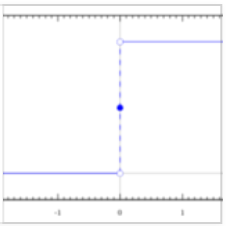
> *There are many kinds of activation functions but we shall only be discussing the few important ones. For more information on activation functions refer the links below:*

Fundamentals of Deep Learning—Activation Functions and When to Use Them?

www.analyticsvidhya.com

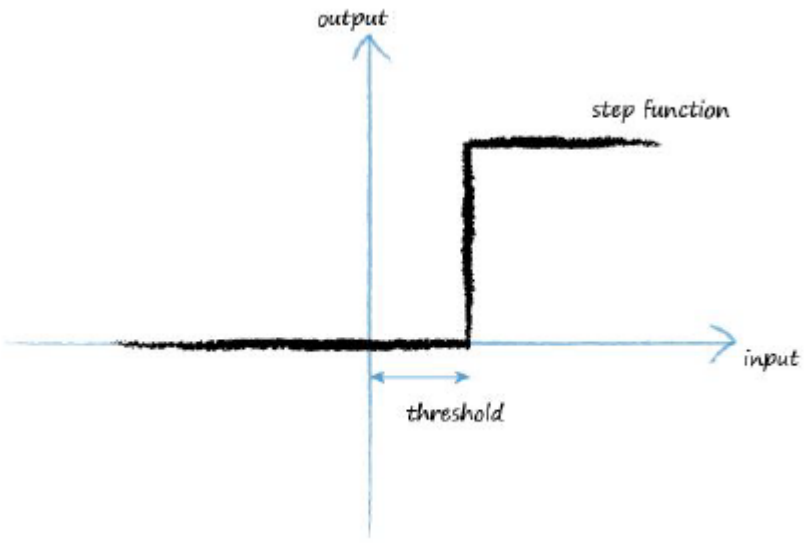
Understanding Activation Functions in Neural Networks

medium.com



- **Step Function**

In an `Step Function` output is zero for low input values, and output jumps once the threshold input is reached.



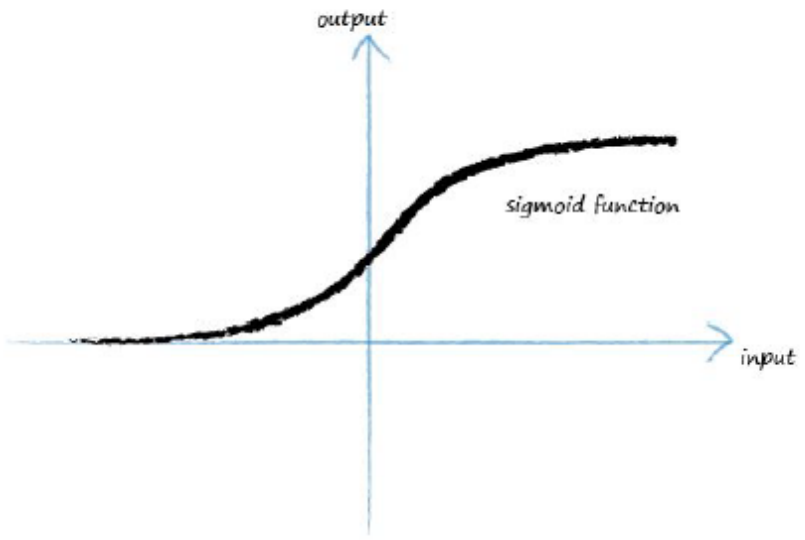
A plot of a Step Function

The Step function works well while creating a binary classifier where we need to say yes or no for a single class. However, it fails in case of a multi-classifier problem.

- **Sigmoid Function**

A sigmoid function is smoother than a step function. It is also called a logistic function and is governed by the formula:

```
y = 1/(1+e^-x)
```



The plot of a Sigmoid Function

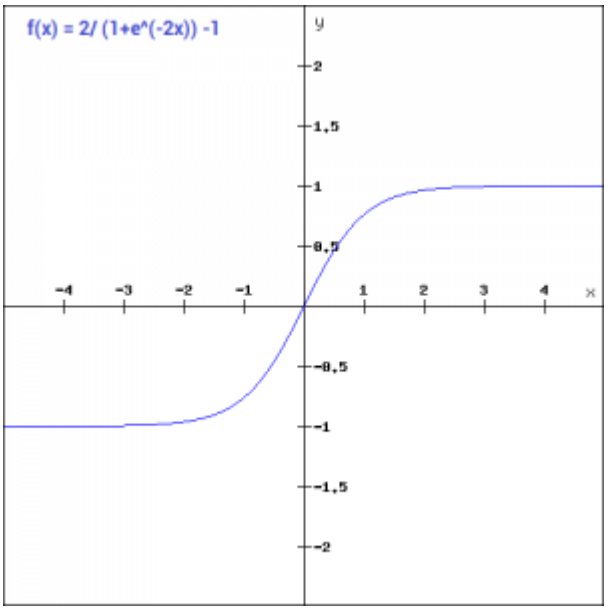
A Sigmoid Function is non-linear, unlike a Step Function. The output of the activation function is always within the range (0,1). Sigmoid functions are **widely used**, but they suffer from a disadvantage. If we look at the Sigmoid Plot, we notice that towards either end of the function, the Y values do not change much on changing X. This implies that the slope/gradient at these regions is going to be very small and this gives rise to **'Vanishing Gradients'** problem. This means the network doesn't learn further or is very slow in learning due to the minimal values of gradients. Nonetheless, sigmoid activation functions are still primarily used in classification problems.

- **Tanh Function**

Tanh function is just a scaled version of the Sigmoid Function but is symmetric over the origin. Tanh function range from -1 to 1. Their formula is as follows:

```
tanh(x)=2sigmoid(2x)-1

tanh(x)=2/(1+e^(-2x)) -1
```



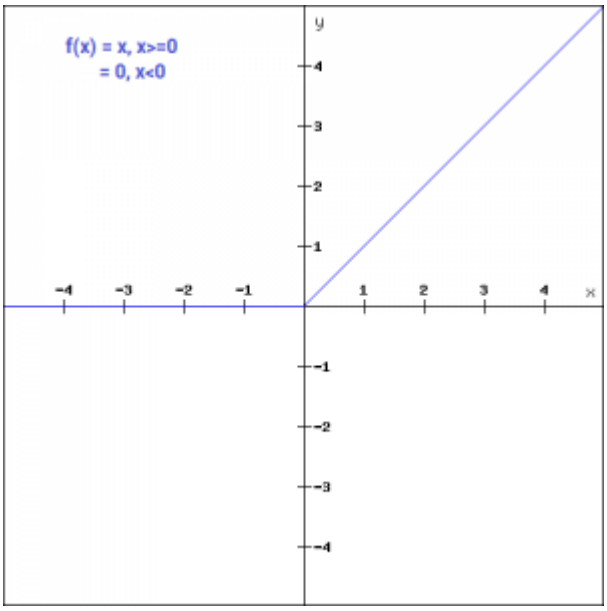
The plot for Tanh function. Source

The gradient of the Tanh function is steeper as compared to that of the sigmoid function, but it also suffers from the vanishing gradient problem. So how do we choose between Sigmoid and Tanh? This depends on the requirement of the gradient in the problem statement. It is also a very popular and widely used activation function.

- **ReLU**

The ReLU or the Rectified linear unit is the most widely used activation function. It is also a non-linear function and can be expressed as :

```
f(x)=max(0,x)
```



The plot of the ReLU function

The primary advantage of using the ReLU function over other activation functions is that it does not fire all the neurons at the same time. This means only a few neurons are activated at a time, thereby making the network sparse and easy to manage. ReLU is also less computationally expensive than Tanh and sigmoid because it involves simple mathematical operations which comes in handy when designing deep neural networks.

However, vanishing gradients problem also affects ReLU function.

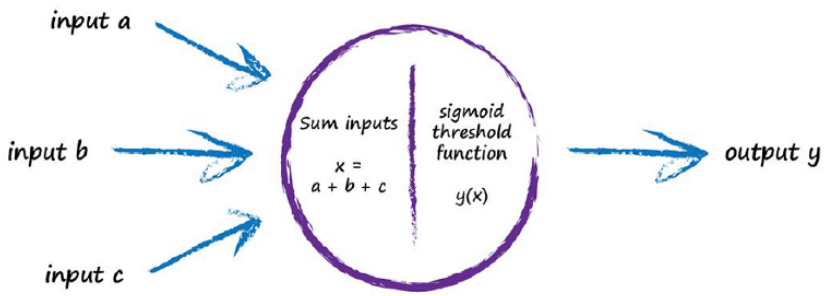
After having a general outline of the significant activation functions, we can choose the activations which best suits the problem at hand. In this article, however, I will stick with the sigmoid activation function for simplicity.

. . .

Modelling an Artificial Neuron

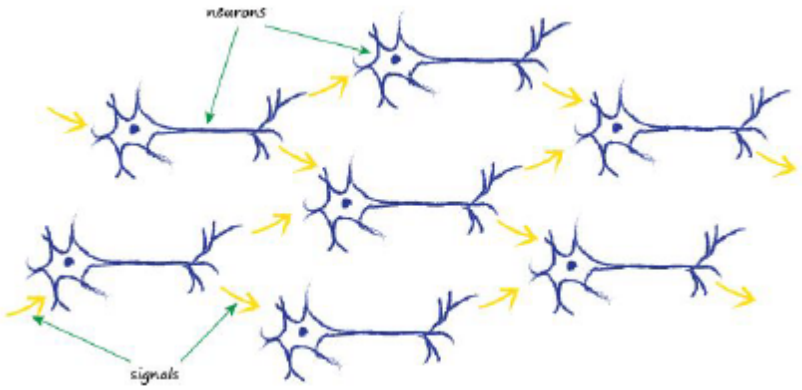
After going over the basics, it's time to model an artificial neuron from scratch. After going through this section, you will not only have a better understanding of the neural network but will also gain a better understanding of its intricacies and structure.

A biological neural takes multiple inputs rather than just one. Since we have more than one input, a logical way would be to sum them up and pass it through an activation function(e.g. Sigmoid Function), which will, in turn, control the output.

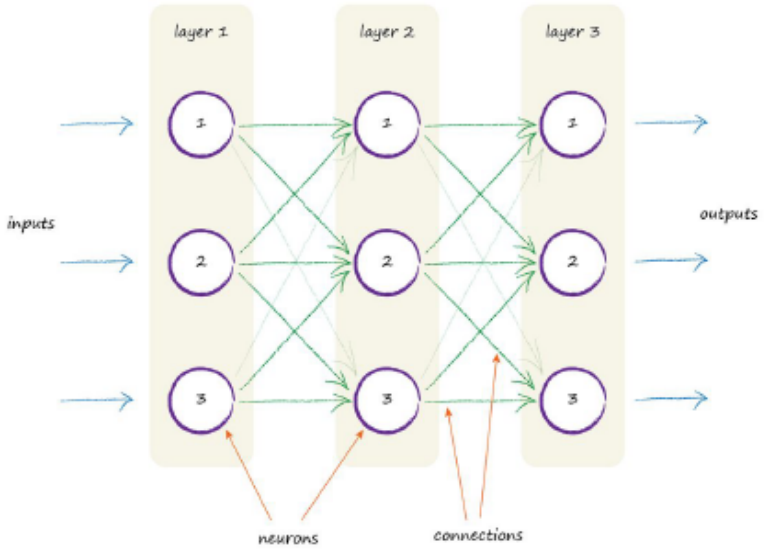


The neuron will only fire if the combined sum of inputs exceeds the threshold

Let us look at a typical scenario of electrical signal conduction in our brain. If we could visualise it, it would be something like this:



The above diagram shows several interconnected neurons. Each neuron takes input from and also outputs signals to more than one neuron at a time. To map this feature into an artificial model is by drawing many layers of neurons, with each layer connected to the preceding and the subsequent layer.



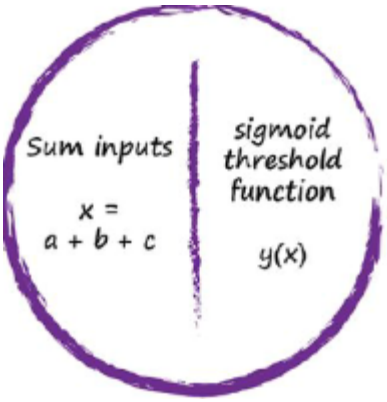
A three layered neuron

The figure above represents

- A 3 layered Artificial neuron model
- Every layer, in turn, has three nodes marked as 1,2 and 3 respectively
- Every node is connected to nodes of the previous and the next layer

The architecture is fine, but which part undergoes the training? Is there a parameter that can be adjusted just like the slope in case a simple classifier?

We can proceed in a neural network by adjusting the strength of connections in between nodes. A node(as seen in previous sections) can be represented as follows:



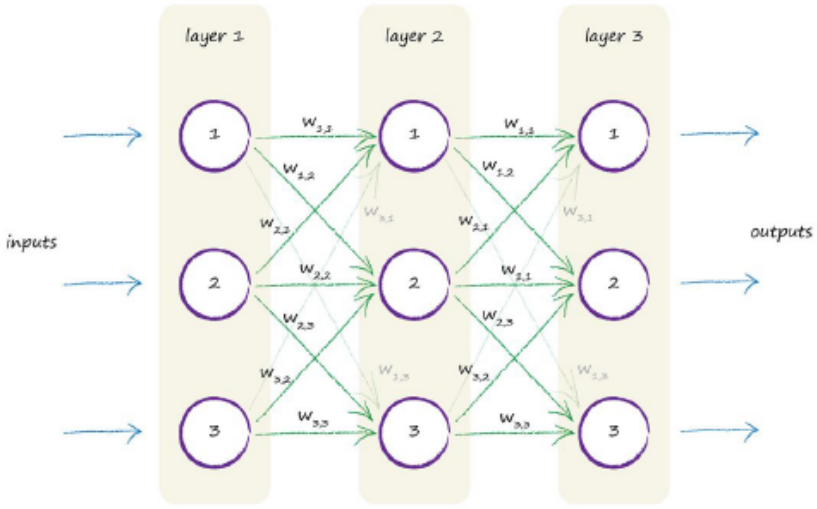
a single node

We can adjust the strength of the connections either through:

- varying the strength of the summation of inputs, Or
- through adjusting the shape of the activation function.

Since the former is a simple solution, let us stick with that.

Now, let us represent the same artificial neural network architecture with weights associated with each connection. The weights are important parameters since a **low weight suppresses a signal while a high weight amplifies it.**



Weights are represented by the symbol '**W**', and the notation is as follows:

$$W_{1,2}$$

weight associated with the signal between node 1 in a layer to node 2 in the next layer

We have shown that each node is connected to every other node in the previous and the next layer. During the learning phase, the extra connections will get de-emphasised in case they are not needed.

This means that during the training phase, as the network learns to improve its outputs by refining the link weights inside the network, some weights become zero or close to zero. Zero or almost zero, weights means those links don't contribute to the network because signals don't pass. A zero weight means the signals are multiplied by zero, which results in zero, so the link is effectively broken.

. . .

Conclusion

In this article ,we learnt about the building blocks of neural networks. We created a simple predictor and classifier from scratch and understood how they form the basis for understanding the neural networks. We also learnt how a neural network is in a way a manifestation of the human brain and visualised it in the same way.

In the next part, we will study about the working of the neural network by simplifying the entire process through a bunch of calculations. We will also simplify the concept of backpropagation and weight updation with the help of simple mathematics. Till then happy learning.