How Do Computers Learn?

Modelling a human brain
through mathematics

TABLE OF CONTENTS

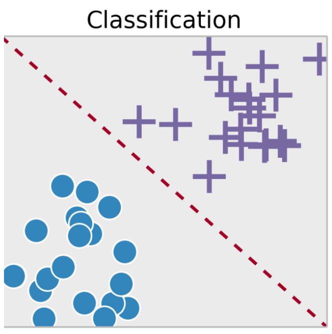
Sr. No.	Title	Page Number
1	Introduction	2
2	Neural Networks as Classifiers	2
3	Adopting the Neuron Model	3
4	Computation	6
5	Generalisations	11
6	Conclusion	14
7	Evaluation	14
8	Bibliography	15

INTRODUCTION

Trying to mimic the way a human brain learns has long been an buzzing area of research in the field of Mathematics and Computer Science. Although it may seem quite complicated, an analogy derived from the architecture and how that connects to logistic functions and calculus makes it particularly intuitive as we shall see in this exploration.

Neural Networks can be used as classifying functions which take in sets of data and classify them as we want them to. For example, if a researcher were to find out the beak length and beak width of two species of birds, a Neural Network could take that data and generalise to a function so that if any new unknown bird's beak height and width were to be inputted in the function, it would classify between the two birds accurately. In the 1950's mathematicians realised that human brains were great at classifying complicated things like identifying whose face belongs to whom or what a cat or dog look like based on physical features. However, recognising and classifying such data would be almost impossible to do mathematically. Therefore, in 1957, Dr. Frank Rosenblatt devised the perceptron which was modelled around the human brain and could learn to classify data for multiple dimensions. It could take in a lot more parameters in the bird example, like wing span, claw length etc. in order to classify even more birds. In this exploration, the mathematics behind a perceptron will be explained along with its real world uses.

Neural Networks as Classification Functions.



A classification function is essentially a function that can classify data into two or more groups based on one or more parameters. Therefore a classification algorithm would be able to classify the species of birds based on various parameters such as beak length, weight, wingspan etc. It works similar to linear regression but the line it is trying to estimate is the decision boundary.[1] A decision boundary is a line(as seen as dotted line in figure 1) that separates the two sets of data in order to classify them. The given image shows an ideal scenario, however, real world data sets can be non-linearly separable which calls for a non-linear decision boundary. Neural Networks do exactly that with complex data.

[1] https://www.cs.princeton.edu/courses/archive/fall08/cos436/Duda/PR_simp/bndrys.htm

ADOPTING THE NEURON MODEL

The human brain contains billions of tiny cells called neurons. These neurons relay information to each other through structures called dendrites and axons. **Neurons excite and send signals** to one another causing us to perform some actions. These connections between neurons **strengthen when the brain is learning something**. For example, when the brain learns something new about math, neuron pathway related to math strengthen, i.e. the connection between the associated neurons strengthens. Thus to make computers learn a task, we can model these neurons mathematically.

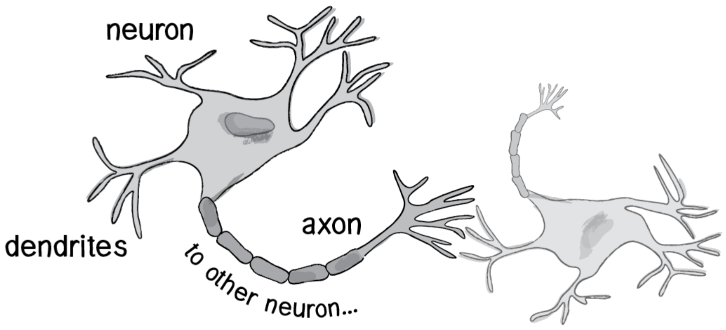


Fig 1.0. An enlarged diagram of a Neuron Connected to another Neuron

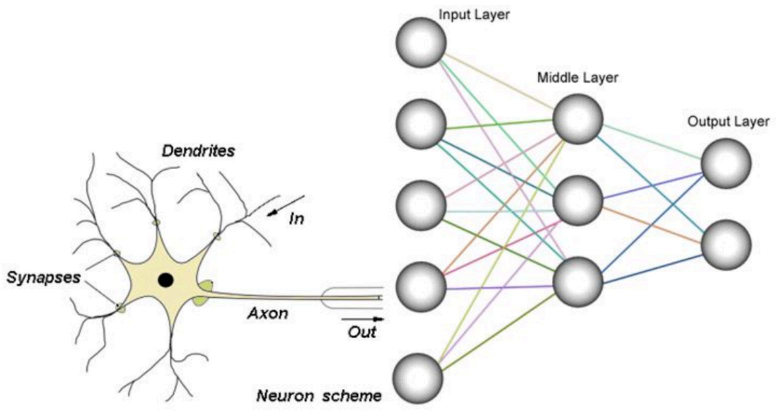


Fig 2.0. This diagram shows shows how neurons might be connected in a Neural Network

We can imagine these Neurons to store some information and perform a simple function. Therefore, our mathematical Neuron will store a number and perform a Mathematical Function ***f***. This function is also called as **Activation Function.**[2] Now we can imagine a network of these connected neurons as shown in Fig 2. An individual neuron takes inputs from whichever neurons it is connected to, on the left and computes an output which it feeds to any neuron it is connected to on the right. As seen in the diagrams, these sets of Neurons can be labelled under one class. The leftmost Neurons are called **Input Layer**; these neurons take in the **input variables**[3] which may either be discrete or continuous; the Output layer is the right most layer which holds the output of the Neural

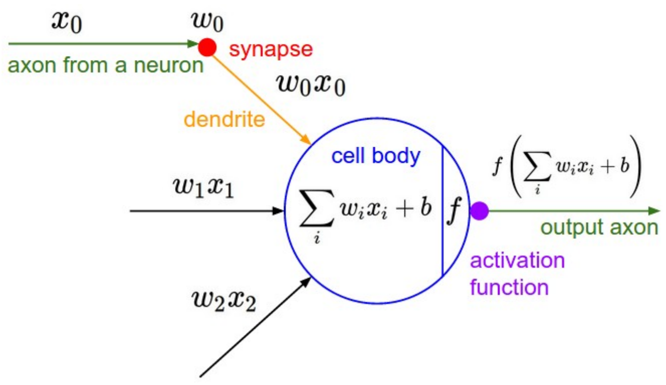


Fig 3.0: Mathematical Model of a Neuron

[2] https://www.techopedia.com/definition/33261/activation-function

[3] (just like *x* in *f(x)* as this Neural Network can be thought of as a complex composite function with more than one variable as input)

Network. The middle or **"hidden layer"** is where the computation happens.

The connections between these Neurons are **actual numerical coefficients or 'weights'** which determine the '**strength**' of the connection and relationship between two consecutive Neurons in different layers. In Fig.3 x_i are the outputs from other neurons which are inputs to the given Neuron. This weighted sum of all the inputs where w_i are the weights or coefficient of each connected neuron x_i , is computed. This weighted sum is then added to a **threshold or 'bias'** (**B**). This weighted sum plus bias is then put through an 'activation function', which relates the computed sum so far to the actual output of the Neuron which determines whether the Neuron will be '**active or inactive.**'[4] This function can be of many different types and will be explored further. A Neural Network learns by adjusting these weights and biases to fit a particular task assigned to it. Essentially, when some neuron associated with a particular function learns something, its connections strengthen, which mathematically translates to its weights increasing in magnitude.

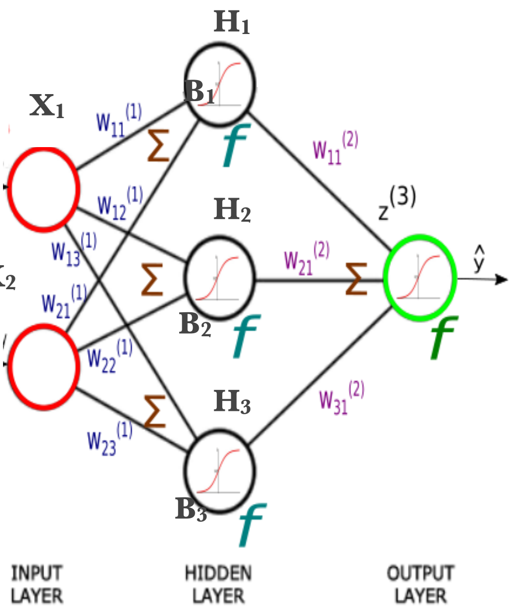


Fig 4.0:: XOR NN model

Now we have a complete model of the **Neural Network (which will now be called NN)**. We can now make a simple NN which can learn to **compute Binary Operations** for the sake of simplicity and understanding.

This network will contain 2 input Neurons. The hidden layer will contain an arbitrary 3 neurons and the output layer will contain 1 Neuron which will output a value between 0 and 1.[5]

Now before we proceed out NN we need to define some notation.

X_u	**—> Input (u is the order of the input)**
H_h	**—> Input (h is the order of the Hidden Neuron)**
$W_{mn}^{(k)}$	**—> Weight connecting the mth neuron of the kth layer[6] to the nth neuron of the (k+1)th layer.**
f	**—> Activation function(function associated with the Neuron)**
$\hat{y}$	**—> Prediction made by Neural Network.**
$B_n^{(k)}$	**—> Bias in the nth neuron of the kth layer.**

[4] 'Active' means that the value output of the Neuron is numerically significant or more positive. If the bias is negative, the weighted sum has to be higher in order for the Neuron to be 'active'. If the bias is positive, the Neuron is likely to be active regardless.

[5] This is known as fuzzy logic. We can assign a representative value to each result that we want. In our case our classification is binary so we can make do with only 1 output neuron. The way we come to these two values will be explained further.

[6] k refers to the numbered layer where the first layer is k = 1.

Computation:

We will first compute the weighted sum which will input to Neuron **H1.**

Let the Weighted Sum of Inputs be Z_{H1}.

Therefore:

$$Z_{H1} = W_{11}^{(1)} \times X_1 + W_{21}^{(1)} \times X_2 + B_1^{(2)}$$

Similarly:

$$Z_{H2} = W_{12}^{(1)} \times X_1 + W_{22}^{(1)} \times X_2 + B_2^{(2)}$$

And so on…

As discussed before, **this weighted sum** is put through an activation function which determines whether the Neuron will fire or not. It has also been established that the range of the output of this function must be between 0 and 1. Therefore, a suitable candidate for this function is the **Sigmoid function.**

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

This function has some desirable properties that we can exploit in order to make our classification algorithm.

If we did not have an activation function, our model would be linear i.e., the two inputs would only be added and multiplied by the weights or coefficients. Then it would become a linear regression model which is not good for classification.

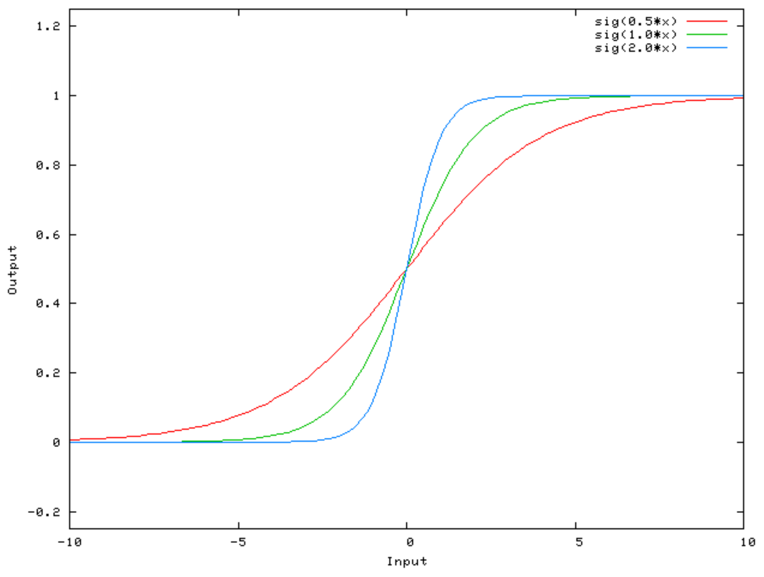


Fig 5.0: Sigmoid function transformations

Moreover, the sigmoid function helps in the 'activation' part of the neuron. If the input is low enough, the output that it has is 0. If the output is high enough, the output quickly converges to 1 as seen in the Fig 5.0. In an untransformed sigmoid function, the point of inflection of the function is at **x = 0**. This means that if the input is negative, its value is below 0.5 and if its value is positive, it is more than 0.5. However, if we increase the weights of the input, the point of inflection of the function become steeper. This holds with our Neuron analogy. If the Neuron learns something associated to it, its weights increase. Thus it becomes 'more sure' of when to fire, thus its decision boundary between 0 and 1 becomes steeper. We can see this in Fig. 5.0 as Sig(2x) has a steeper boundary between 0 and 1. The Bias(B) however, has the effect of shifting this point of infection on a horizontal scale, so a positive bias will shift the point of inflection to the left and vice versa. This again holds with the Neuron analogy as the bias decided at what ‘**threshold**’ or extent along the x axis will the Neuron fire.

Therefore, the output of Neuron $\mathbf{H_h}$ would be $\boldsymbol{\sigma}(\mathbf{Z_{kh}})$.

The final output of the Neuron would be called $\mathbf{H_h^k}$ where k is the layer number and h is the Neuron number in that layer.

$$H_h^{(k+1)} = \sigma(Z_h^{(k+1)}) = \sigma(W_{mn}^{(k)} \times X_m + W_{mn}^{(k)} \times X_m + B_1^{(k)})$$

Similarly, the output can be computed to be:

$$\hat{y} = \sigma(Z_1^{(3)}) = \sigma((W_{1,1}^{(2)} \times H_1^2) + (W_{2,1}^{(2)} \times H_2^2) + (W_{3,1}^{(2)} \times H_3^2) + B_1^{(3)})$$

To generalize further, the weighted sum acts like the dot product between the Input vector and the Weight matrix.

$$w_{k,\ (k+1)} = \begin{bmatrix} W_{1,1}^{(k)} & W_{2,1}^{(k)} \cdots & W_{v,1}^{(k)} \\ W_{1,2}^{(k)} & W_{2,2}^{(k)} \cdots & W_{v,2}^{(k)} \\ & \vdots & \\ W_{1,u}^{(k)} & W_{2,u}^{(k)} \cdots & W_{v,u}^{(k)} \end{bmatrix}$$

Where **v** is the size of the layer **k** and **u** is the size of the layer **(k+1)**

Here *w* is the weight matrix between the layers *k* and *k+1.* The notations k at the top of each weight notation can now me omitted as it is expressed in the start of the weight matrix.

We can now imagine the layer of Neurons which are acting as the input to the next layer of Neurons as a vector **N.**

$$n_k = \begin{bmatrix} H_1 \\ H_2 \\ \vdots \\ H_v \end{bmatrix}$$

Where **v** is the size of layer **k**

Finally we have need to represent the biases of each Neuron except for the input neurons. Therefore:[7] [8]

$$b_{k+1} = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_u \end{bmatrix}$$

where u is the size of layer k+1.

[7] The biases are only added to neurons other than the input Neurons. Input Neurons are just for the sake of continuity, however, these ‘Neurons’ are just numerical values.

[8] Here (k+1) one is used instead of k because the Bias is always linked to the Neurons that are taking the input from other Neurons rather than the Neurons acting as the input.

$$n_{k+1} = \sigma\left(\begin{bmatrix} W_{1,1}^{(k)} & W_{2,1}^{(k)} & \dots & W_{v,1}^{(k)} \\ W_{1,2}^{(k)} & W_{2,2}^{(k)} & \dots & W_{v,2}^{(k)} \\ & \vdots & & \\ W_{1,u}^{(k)} & W_{2,u}^{(k)} & \dots & W_{v,u}^{(k)} \end{bmatrix} \cdot \begin{bmatrix} H_1 \\ H_2 \\ \vdots \\ H_u \end{bmatrix} + \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_u \end{bmatrix}\right)$$

Now that we have our notation down, the Computation of the output of neurons in each layer can be concisely written as the following:

$$n_{k+1} = \sigma((n_k \cdot w_{k,\ (k+1)}) + b_{k+1})$$

This General Formula is applicable to all layers of the Neural Network. Recalling our Binary Operator Neural Network, we can now make predictions on the answer of the Operator, given two binary inputs.

We will make an XOR Binary Operator for the sake of simplicity, in which, if both the inputs are the same, it will output 0 and if the either of them are different, it will output a 1.[9]

Even such a simple task cannot be solved using a classical linear classifier! Here's why.

In figure 6.0, there is no possible straight line that can divide the orange and blue dots into two regions. Therefore a linear function cannot classify this data set. A Network Network, however, can compute a **non-linear decision boundary**[10] which is its speciality.

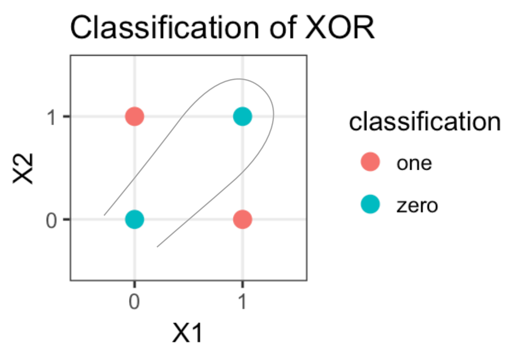


Fig 6.0: Hypothetical Non-linear Decision Problem

Now, in the case of our NN Classifier, it will perform the following Operations for each given input vector **X,** containing x1 and x2.

To compute Second, Hidden Layer…

$$n_2 = \sigma((X \cdot w_{1,2}) + b_2)$$

To compute Third, Output Layer…

$$\hat{y} = n_3 = \sigma((n_2 \cdot w_{2,3}) + b_3)$$

Logic for XOR problem		
x1	*x2*	$y = \overline{\mathbf{x1} \oplus \mathbf{x2}}$
0	0	0
1	0	1
0	1	1
1	1	0

Fig 7.0: XOR Table

[9] You may be asking that this is an inherently simple task to do manually. However, this is a simple case to showcase the classifying capability of a Neural Network. Also note that we are not defining the operator. We are only 'showing' the Neural Network examples and it will learn how to do the task of the operator.

[10] Recall from page 1

Thus our NN can now make predictions given input vector ***X*** which contains **x1** and **x2.** Our output should be **y**, pertaining to the table in figure 7.0. However, we have not assigned any values to the weights and biases. This is where the **learning** part comes in. We have to assign weights and biases to the NN so that it can optimally predict the answer **ŷ**. To assign these weights, we use a method called **Back-propagation and Gradient Descent which uses multivariable calculus to optimize for these weights and biases.**

BackPropagation

Since it was established that this method uses multivariable optimization, we must also establish an objective or cost function. Since we have a set of input and output values, we can compare the prediction of our NN with the expected output which is given in Figure 7.0. Thus we can measure how 'well' or how 'poorly' our NN predicts the outcome by calculating the distance between between the Expected output **y** and **ŷ** for given input **X**. We can call this difference the 'Error'. However, simply calculating the difference between the two will not give us an accurate measure of performance, as trying to minimize this cost function may cause the difference to be negative which defeats the purpose. Hence we can square this error to give a positive parabolic curve which can be minimized efficiently. It also has the additional property of amplifying large errors more than small ones.

$$cost = (\hat{y} - y)^2$$

Now, the parameters that we need to optimize are those weights and biases, given the inputs and expected outputs. We do this by computing partial derivatives of the cost function with each weight and bias. To do this, we must write the cost function in terms of these weights and biases.

Given:

$$cost = (\hat{y} - y)^2$$

$$\hat{y} = \sigma((\sigma((X \,.\, w_{1,2}) + b_2) \,.\, w_{2,3}) + b_3)$$

Substituting:

$$cost = (\sigma((\sigma((X \,.\, w_{1,2}) + b_2) \,.\, w_{2,3}) + b_3) - y)^2$$

Just by looking at the above equation, finding the partial derivative of each weight is difficult. Since the cost function is just a large composite function, we can use the chain rule to find these derivatives. We can recall the NN graph in Figure 8.0 as an analogy to find the derivatives.

We can first compute the derivative of the cost function with respect to **ŷ:**

$$\frac{d(cost)}{d\hat{y}} = 2(\hat{y} - y) \qquad \text{........ } \textbf{Eq. 1}$$

Since the output **ŷ** is the the sigmoid of the weighted sum+bias, we can apply chain rule to get:

$$\frac{d(cost)}{d(Z_1^3)} = 2(\hat{y} - y) \;.\; \frac{d(\sigma(d(Z_1^3)))}{d(Z_1^3)} \qquad \text{........ } \textbf{Eq. 2}$$

To proceed further, we must find the derivative of a sigmoid function.

By applying the chain rule along with the power rule, we get:

$$\frac{d(\sigma(x))}{dx} = -\frac{1}{(1+e^{-x})^2} \;.\; \frac{d(1+e^{-x})}{d(x)} \qquad \text{........ } \textbf{Eq. 3}$$

Applying chain rule further…

$$= -\frac{1}{(1+e^{-x})^2} \;.\; (0 - e^{-x}) \;.\; \frac{d(x)}{d(x)} \qquad \text{........ } \textbf{Eq. 4}$$

Rearranging…

$$= \frac{1}{(1+e^{-x})} - \frac{1}{(1+e^{-x})^2} \qquad \text{........ } \textbf{Eq. 5}$$

$$= \sigma(x) - (\sigma(x))^2$$

$$\boxed{\therefore \; \frac{d(\sigma(x))}{dx} = \sigma(x) \; (1 - (\sigma(x))} \qquad \text{........ } \textbf{Eq. 6}$$

Thus we can compute:

$$\boxed{\frac{d(cost)}{d(Z_1^3)} = 2(\hat{y} - y) \;.\; \sigma(Z_1^3) \; (1 - (\sigma(Z_1^3))} \qquad \text{........ } \textbf{Eq. 7}$$

Thus, note that our objective is to find the derivative of the cost function with respect to various weights and biases. Now that we have a derivative of the same with respect to the weighted sum of the final output Neuron, finding the derivative of this weighted sum **Z** with respect to the weight becomes easy. Referring back to Figure 4.0 we can visually see this chain of derivatives going backward from the output Neuron. Hence this process is known as Backpropagation. We know that:

$$when \;\; y = f(x) + z$$

$$\frac{dy}{dx} = \frac{d(f(x)+z)}{dx} = \frac{d(f(x))}{dx} + 0$$

Therefore we can take partial derivatives of each weight.

Thus:

$$\frac{d(cost)}{d(\ W_{1,1}{}^{(2)})} = 2(\hat{y} - y)\ .\ \sigma(Z_1^{(3)})\ (1\ -\ (\sigma(Z_1^{(3)}))\ .\ \frac{d(Z_1^{(3)})}{d(\ W_{1,1}{}^{(2)})}$$ .... **Eq. 8**

$$since,\ Z_1^3\ =\ ...\ (Z_1^{(2)}\ \times\ W_{1,1}{}^{(2)})\ ...\ +\ B_1^{(3)}$$ recall from page 4

$$\frac{d(cost)}{d(\ W_{1,1}{}^{(2)})} = 2(\hat{y} - y)\ .\ \sigma(Z_1^3)\ (1\ -\ (\sigma(Z_1^3))\ .\ Z_1^{(2)}$$ .... **Eq. 9**

GENERALISATIONS

Thus we can **generalize** a formula for finding the derivative of the cost with respect to a particular weight.

$$\boxed{\frac{d(cost)}{d(\ W_{m,n}{}^{(k)})} = \frac{d(cost)}{d(H_m^{(k+1)})}\ .\ \sigma(Z_n^{(k+1)})\ (1\ -\ (\sigma(Z_n^{(k+1)}))\ .\ Z_m^{(k)}}$$ .... **Eq. 10 (Generalization)**

This makes back-propagation very neat and simple to understand. Now, once we have the derivatives with respect to all the Neuron Outputs, we can compute the derivative with respect to any weight or bias. Differentiating with respect to bias also becomes easier. Since the bias is added at the end of the weighted sum, its partial derivative will be the partial derivative with respect to the Weighted Sum of the Neuron it is associated with, into 1.

$$\boxed{\begin{aligned}\frac{d(cost)}{d(\ B_m{}^{(k)})} &= \frac{d(cost)}{d(H_m^{(k)})}\ .\ \sigma(Z_m^{(k)})\ (1\ -\ (\sigma(Z_m^{(k)}))\ .\ \frac{d(Z_m^{(k)})}{d(\ B_m{}^{(k)})}\\ & \frac{d(cost)}{d(H_m^{(k)})}\ .\ \sigma(Z_m^{(k)})\ (1\ -\ (\sigma(Z_m^{(k)}))\ .\ 1\end{aligned}}$$ .... **Eq. 11 (Generalization)**

Now, we can calculate the derivative with respect to the weighted sum of each Neuron, given the partial derivative of Neurons which it acts as input for. Note that this is a complex implementation of the chain rule. Since the Neural Network is a multilayered composite function.

$$\boxed{\frac{d(Z_n^{(k+1)})}{d(Z_m^{(k)})} = \frac{d(W_{m.n}{}^{(k)}\ .\ \sigma(Z_m^{(k)}))}{d(Z_m^{(k)})} = W_{m.n}{}^{(k)}\ .\ \sigma(Z_m^{(k)})\ (1\ -\ (\sigma(Z_m^{(k)}))}$$ .... **Eq. 12 (Generalization)**

Now that we know how to find the derivative of the cost with respect to all the weights, biases and Neurons the Neural Network can finally learn. By learning, the Neural Network is basically finding the optimum weights and biases to find the **minimum cost or deviation from the expected value**. So how does finding the derivative help? Now that we know the slope in which the point of the multidimensional cost function is, we can move our point closer to the minimum by moving the specific weights, starting from a random point, towards the minimum. For example, if we have a parabola **$y = x^2$.** To find the global minimum, using this method, a random x is taken. Say **$x = 5$**. Now we know the derivative of **y** with respect to **x** would be be **2x** which is equal to **10.** Hence we move the **x value** in a direction **opposite to the positive slope** in order to reach the x value at the minimum which is **$x = 0$.** We can define this movement by the follow equation:

$$\Delta x = -\frac{dy}{dx} \times \varphi \quad \textit{where } \varphi \textit{ is the learning or rate of descent} \text{ .... } \textbf{Eq. 13}$$

Here Δ**x** is the movement of **x**. This Δ**x** is added to the existing value of x to get closer to the minimum. **This is called Stochastic Gradient Descent.**[11] The learning rate ϕ can determine by how much proportion Δ**x** moves towards the minimum. If learning rate is high, the algorithm will converge to the estimated minimum faster but it will lose accuracy. If the learning rate is low, gradient descent will take time to converge to an estimated minimum but it will gain accuracy.

The same applies to each weight and bias in the Neural network.

Now that we have all the necessary equations, we can train our Neural Network starting with random weights and biases.

Thus, to summarize how a Neural Network learns, it starts off by assigning random parameters(weights(coefficients) and biases) which produces a random prediction **$\hat{y}$** which is compared to the expected prediction for the given input. The derivative of this error is computed with respect to all the weights and biases. These derivatives are used to descend down the gradient of the defined cost function to find its minimum, thus optimizing our model of a brain which can classify datasets based on their traits. This process is repeated many times with different known data-points till the error is negligible, meaning the Network has reached its minimum. This is demonstrated in the XOR Neural Network that is defined thoroughly above. The functions and generalizations were put into a simulator to produce the following results.

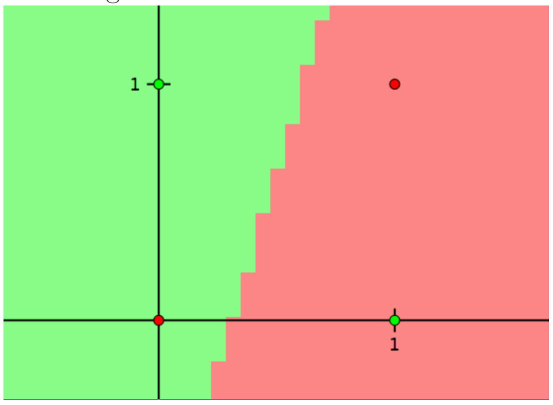


Fig 9.0: Untrained Neural Network with inaccurate Predictions. Note that the red dot is in the green region and vice versa.

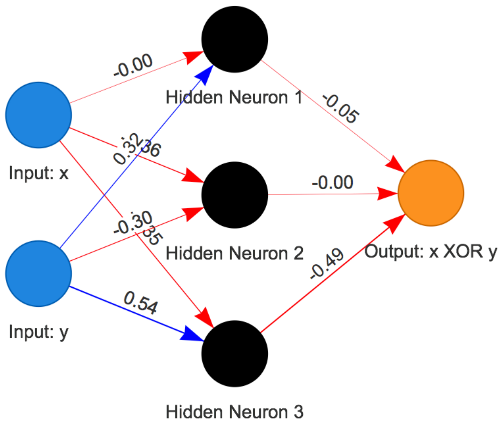


Fig 8.0: Untrained Neural Network

[11] https://metacademy.org/graphs/concepts/stochastic_gradient_descent

In the following diagrams, we cans see the randomly assigned weights and biases produce an inaccurate results.

However, now we shall see, after countless iterations, the network learns to classify the dots in an XOR manner via our BackPropagation Model.

Here we can see the Neural network getting optimized through the laid out BackPropagation Algorithm.

The cost function reduced with each step. The weights and biases also update themselves. Fig.10.0 clearly depicts the Root Mean Square Error(RMSE) which is just the root of proposed cost function reducing with each iteration till it reaches a negligible error of ~0.03. Note that the Error starts from 0.50 as the weights and biases are assigned randomly. As the model is tested against the XOR table and the derivative of weights and biases are calculated, the back-propagation algorithm moves the weights and biases towards the minimum of the cost function by **Eq. 13**.

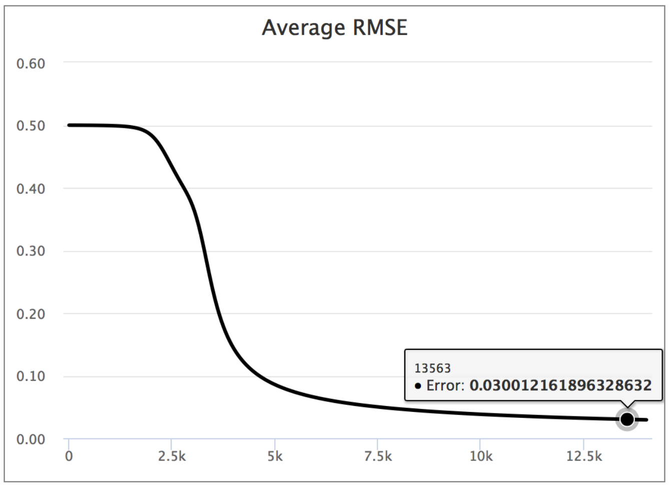


Fig 10.0: Error(cost Function) vs number of iterations of the algorithm

In **Fig 11.0** which represents the optimized Neural Network, the thicker blue arrows represent a strong positive relationship between the Neurons and the thick red arrows represent strong, negative relationships between the Neurons. This is the result after around 13000 iterations of this algorithm. Modern computers are capable of doing such tasks in seconds.

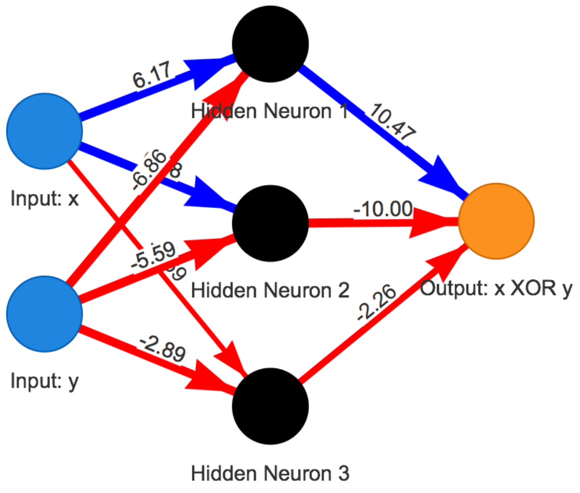


Fig 11.0: Trained Neural Network

Thus our classifier draws a decision boundary to differentiate the two sets of data as shown in Fig. 12.0. The red area on the graph represents the region where $\hat{\mathbf{y}}$ is less than 0.5 and the green area shows the region where $\hat{\mathbf{y}}$ is more than 0.5. Thus the red dots which are an expected value of 0 fall right in the centre of the red region and the same for the green points which represent a value of 1. The boundary between these two regions is known as the decision boundary. Fig. 13.0 shows these results numerically.

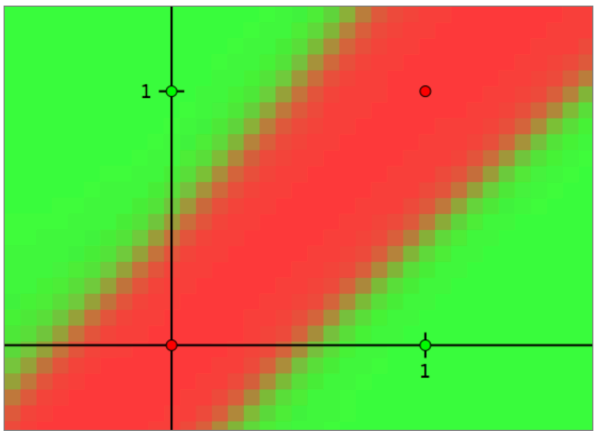
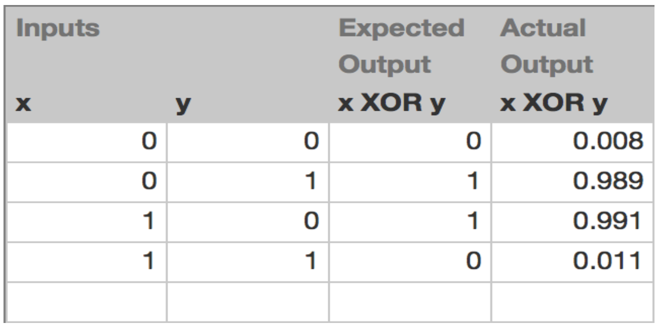


Fig 12.0: Trained Decision Boundary



Inputs		Expected Output	Actual Output
x	y	x XOR y	x XOR y
0	0	0	0.008
0	1	1	0.989
1	0	1	0.991
1	1	0	0.011

Fig 13.0: These are final predictions

CONCLUSION

Initially, an analogy from human brains was drawn to make a mathematical model of Interconnected Neurons called Neural Networks. Then, the necessary notion was laid out in order to make sense out of this model. This was one of the biggest challenges I faced while writing this paper. There is lack of a standard agreed upon notation on Neural networks owing to their recent development and popularization. After laying down the basic underlying equations, a generalized concise vector matrix formula was derived which makes it easier to compute and to read these Networks. Now that the basic, universal and generalized model was established it was time for the model to learn a given task. The XOR problem was chosen as it is a popular classification problem which seems very simple at first but, owing to the non-linear separation of the numbers, classifying these using simple linear classification methods would not yield results. The Neural Network model managed to make a non-linear 'dual' decision boundary with its many layers of neurons and managed to classify the data with astounding accuracy.

EVALUATION

Apart from this simple problem, the generalizations made for the model of a Neural Network means that it can perform highly complex tasks. One such task is Image and Handwriting Recognition. In this, each pixel value of the image is one input Neuron and there are 10 output Neurons which indicate which digit is in the image. For calculating the error for this network, the errors for each output Neuron compared to the expected value are summed up. Since each error is added, the derivative of errors with respect to individual weight and bias are added. This application can be expired further. The genius behind Neural Networks is they are self learning. They can adapt to any dataset as classifiers. They work by combining various decision boundaries of each individual Neuron to form one complex decision boundary that works on the entire dataset after optimization.

One limitation of Neural Networks is that they require a lot of iterations and training data in order to work properly. Moreover, depending on how the weights and biases of the network are selected, Network tends to get stuck on a local minimum which defeats the whole purpose. I experienced this myself when I tried to make a program to execute the Neural Network. It would get stuck on a local minimum which would produce less accurate results. Another limitation is that Neural Networks cannot work on data outside of the domain of the training dataset i.e. it cannot extrapolate. Therefore, they are given a varied dataset when training to recognize images or complex weather patterns etc. so that any scenario falls within its domain.

Neural Networks may sound and look complicated because they are known to be 'self learning', however, everything boils down to multivariable calculus and optimization and some lengthy algebra. Owing to such lengthy computations and derivations, these tasks can only be performs on advanced computers.

BIBLIOGRAPHY

1. **Decision Boundary definition**

   https://www.cs.princeton.edu/courses/archive/fall08/cos436/Duda/PR_simp/bndrys.htm

2. **Activation Function definition**

   https://www.techopedia.com/definition/33261/activation-function

3. **Theory and Conceptual Understanding:**

   https://www.tensorflow.org/tutorials/keras/basic_classification

4. **Neural Network Simulation:**

   https://lecture-demo.ira.uka.de/neural-network-demo/

5. **Figure 1.0**

   https://www.teco.edu/~albrecht/neuro/html/node7.html

6. **Figure 2.0**

   http://sigmacamp.org/2018/semilabs/ai

7. **Figure 3.0**

   http://cs231n.github.io/neural-networks-1

8. **Figure 6.0**

   https://jarvmiller.github.io/2017/10/14/neural-nets-pt1/

9. **Concepts:**

   http://www.youtube.com/watch?v=aircAruvnKk