Efficiency Per Marginal Intermediary Node in a Simple Artificial Neural Network

Abdullah Binladin

April 2019

Abstract

This paper examines the correlation between the number of neurons in a simple artificial neural network and the efficiency of the neural network. "Efficiency" is defined by how certain the artificial neural network is of its results after some number of training cycles t (in the case of this paper, 250 training cycles). The neural network is trained to predict the output of an XOR gate, with two possible inputs 0 or 1. The results show a logarithmic relationship between the number of neurons in the intermediary layer and the certainty of neural network's prediction. The algorithm fails to function at all if the number of neurons in the intermediary layer is smaller than the number of input neurons.

Contents

1 Research Methods and Objectives

1.1 Justification

It is no secret that Artificial Intelligence is among the most prominent fields of Computer Science today. It is also no secret that big media and tech companies collect your browsing data to train AI (Artificial Intelligence) algorithms to serve various purposes (usually to personalize advertisements to you, the user). For instance, Netflix uses your viewing history to train an AI to predict what movie or T.V. Series you may be interested in watching next.

An arguably more nefarious example may be in the case of the two online marketing giants, Facebook and Google. At the forefront of A.I. research, Facebook and Google are known to log their users browsing history to use or sell it as training data for Artificial Intelligence Algorithms. The A.I in question is then used to display advertisements to the user that the AI predicts the user would be likely to click on. Both firms have been criticized for their allegedly morally dubious business practices regarding how they respect their users' right to privacy. In 2018, Facebook CEO Mark Zuckerberg was required to explain to the U.S. Congress how precisely facebook used their user's data. The interview was broadcast live and lasted an entire eight hours.

While the exact artificial neural network algorithm any given media or tech company uses is an extremely closely guarded trade secret - the tech equivalent of Coca Cola's recipe- They all generally the same abstract subcomponents, like how all sodas are some combination of fizzy water, flavouring and colour.

As artificial intelligence is so pervasive in the modern world, it is then relevant to understand how it is that AI algorithms are built and how they function. Thus, this paper assumes that the reader is comfortable with multivariate calculus and is somewhat familiar with linear algebra.

1.2 Objectives

The objective of this research experiment is to model the relationship between the efficiency of a Simple Artificial Neural Network and the number of neurons in its intermediate hidden layer and, if it is possible, to find the optimal number of neurons for this experiment.

As the network used will be a simple artificial neural network, the neural network in this experiment will only have three layers of neurons:

- An input layer of two nodes
- A single hidden intermediate layer of variable size
- An output layer of a single node

The neural network will attempt to predict the expected output of a logical XOR (exclusive OR) gate. An XOR gate is defined as such: an output is true if and only if either of the inputs is true. See Figure 1 for a truth table. As a boolean algebraic expression, it is defined as in Equation 1.

$$\text{A XOR B} : A \wedge B \vee \neg(A \wedge B) \tag{1}$$

I am uncertain as to how an additional node in the hidden layer will affect computational runtime speed. I expect that no matter the number of nodes, the runtime will increase at a factor of $O(n^2)$ such that n is the number of neurons in the intermediate layer. It is also plausible that it could be some form of $O(nlog(n))$. Admittedly I am stretching the definition of Bachmann-Landau notation but it adequately conveys my expectations for the experiment.

XOR Gate		
Input 1	Input 2	Output
0	0	0
1	0	1
0	1	1
1	1	0

Figure 1: An XOR gate in table format

1.3 Methods

Each neuron in the input layer will represent either of the possible inputs of the XOR gate. The output layer will have one neuron containing a number

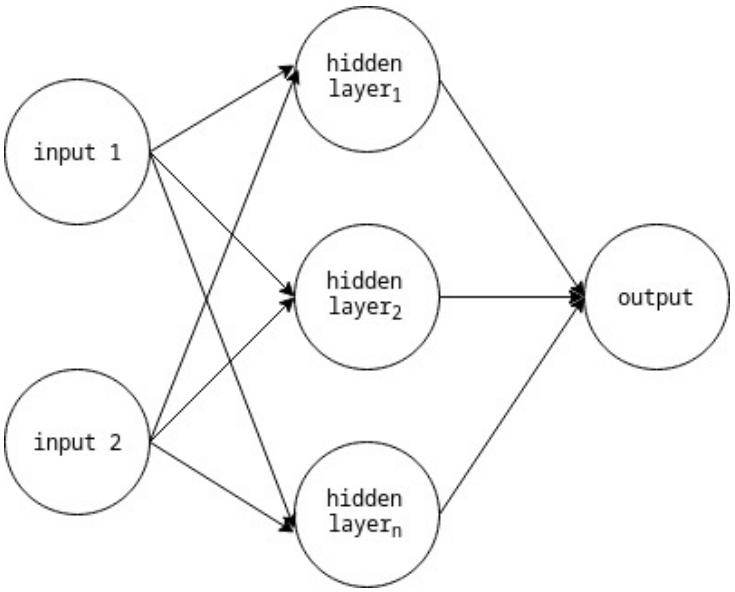

Figure 2: Model of a simple artificial neural network with a hidden layer of length n

between 0 and 1. This number represents the neural network's certainty level.

- The more certain the neural network is that the output should be 1, the closer the output neuron will be to 1
- The more certain the neural network is that the output should be 0, the closer the output neuron will be to 0

The research experiment will be run in a Python 3.7.3 script as Python is both easy to read and is the industry standard for A.I. development. The simple neural network can be modeled as described in Figure 2. The arrows represent the synaptic weights between nodes; n indicates the real coordinate space the intermediate vector lies in.

The python algorithm for building the artificial neural network can be found in 6.1. Upon attempting the experiment I quickly realized that the neural network generated too much data to organize by hand, so I wrote DataCollector.py (Section 6.2)to store and organize the results of the experiment.

Each layer is written as an array of neurons, and each neuron has a corresponding array of synaptic weights. Python does not natively support arrays; instead, it utilizes (Linked) Lists as its main method for aggregating data. Instead, I will be using the numpy Python package.

The independent variable will be the length of the hidden layer. I will begin with a hidden layer of length 1 and increment its length by 1 in each stage of the experiment, up to a final value n.

The dependent variable will be how certain the neural network is of its predicted output. The output neuron denoted by Y_o will contain a value such that $0 < Y_o < 1$; This will correspond to the actual output of an XOR gate, which is either 0 or 1. The closer the output of the neural network is to either 1 or 0, the more certain the neural network is that its output is correct. It should be mentioned that because each neuron is passed through the sigmoid function $\sigma(x)$ (As described in section 2.2), the predicted output will never be equal to 1 or 0; thus it will never be fully certain.

The control variables will be the number of times the neural network is trained, and the number of neurons in the intermediate hidden layer. The former is crucial because the more often a neural network is trained, the more accurate the output becomes. It is only logical, then, to conclude that a change in the number of times a neural network is trained will skew the results of the experiment. The latter's significance is similar; The objective of this experiment is to analyze the effect of an additional neuron on the efficiency of the algorithm- any additional layers would muddy an individual neurons effect on the final predicted output.

All of the data collected is quantitative- there's no qualitative data to collect.

The neural network will be a very basic, barebones, 'vanilla' algorithm; I will not be using any form of deep learning implementations, no optimized initialization of synaptic weights and biases, and the gradient descent algorithm will not be stochastic.

2 Literature Review

2.1 Definition of an Artificial Neural Network

An artificial neural network is a mathematical model loosely based off of how the human brain works. The concept is as follows: a set of layers of

neurons are interlinked via synapses. If a neuron is close to being 'correct', the synapse it's connected to 'fires' and the synaptic weight between them is strengthened. While the latest developments in A.I. have allowed for more complex and more accurate initializations of neurons and synaptic weights, the standard method is to assign each neuron and synaptic weight some random floating-point value between 0 and 1.

An artificial neural network algorithm is created via two processes: building and training Grey (2017). The building process does the actual strengthening or weakening of the synaptic weights between neurons. The training process calculates the error and the correction. These two processes are then run sequentially many hundreds or thousands of times, causing the algorithm to become a little more accurate each iteration.

At no point does either the human, the training processor or the building processor understand precisely what the artificial neural network algorithm actually is or how it arrives at its conclusions; only that with each iteration of training and building, the algorithm becomes marginally more accurate. It is as nebulous and abstract as it sounds.

2.2 Neurons/Nodes

As Sanderson (2017a) says, a *neuron* (also known as a *node*)is simply an element of an array (or a component of a vector, from a mathematical perspective). It contains a floating-point number between 0 and 1. This number is called the neuron's *activiation*. It is this number that represents how 'correct' a neuron is, as is referred to in section 2.1. Each neuron is connected to every neuron in the adjacent layer via synaptic weights.

To calculate the value of a neuron, a linear combination is taken of each connected neuron z of the preceding layer l and its corresponding weight Sharma (2017). In other words, its value is the dot product of each of those neuron layers and their weight matricies:

$$z = \sum_{i=1}^{n} (z^{(L-1)} \cdot w_{jk}) \tag{2}$$

However, this results in a value of $-\infty < Y < \infty$. This, however, does not fit our model for a neuron. There is no upper or lower limit, so we can't calculate how activated a given neuron Y is- there is no frame of reference. Thus, it is necessary to 'squeeze' the value of Y into some value between

0 and 1. This is achieved by passing it into what is called the *Activation function*.Sharma (2017).

There are many activation functions that exist. Some examples are the $RelU(x)$ function and the $tanh$ function. For the purpose of this experiment I will be using the sigmoid function, also known as the logistic function. It is the classic neural network Sanderson (2017b) defined as in Equation 3:

$$Y = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{3}$$

As can be seen in Figure 3, the graph of the sigmoid function has two horizontal asymptotes: one at $y = 1$ and one at $y = 0$, which also fits conveniently with our training goals. No matter what value z is, $\sigma(Y)$ will always be between 1 and 0.

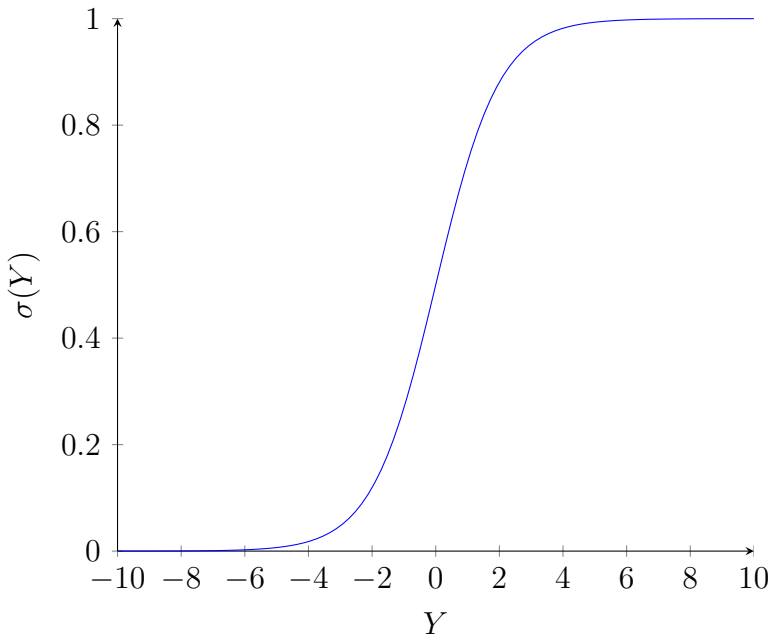

Figure 3: Graph of $\sigma(Y)$

2.3 Synaptic Weights

A *Synaptic Weight* is a ratio representing the strength of the connection between any two neurons Shamdasani (2017). As shown in Figure 2, every

neuron in any given layer is connected to every single other neuron in the layer preceding it. In terms of Computer Science, a set of synaptic weights is functionally identical to a layer of neurons- it is treated as a vector of floating point numbers. The only major difference is that no activation function is applied to a synaptic weight.

The synaptic weights are the main driving force behind an artificial neural network algorithm. With each iteration of the building/training cycle, the synaptic weight is adjusted depending on the severity of the error of that cycle. The more 'incorrect' a synaptic weight is, the more it is adjusted in a given cycle.

2.4 Layers

In section 2.2, I referred to a neuron being an element of an array. In the context of artificial neural networks, this array comes in the form of a *layer*. An artificial neural network is essentially a collection of neurons linked via synaptic weights. The neurons are organized into one of three different categories of neurons. These categories are in the form of layers. These categories are:

- The input layer
- The hidden intermediate layer(s)
- The output layer

All neurons in all three layers are structurally identical- a neuron is simply a variable that holds a floating-point number between 0 and 1. The difference is in what they do: The neurons in the input layer are immutable as it they contain the values of the input, the intermediary hidden layer contains neurons whose values are constantly adjusting and rebalancing depending on the output of the algorithm and the application of gradient descent, and the output layer simply contains the predicted output of a neural network after a training cycle

2.4.1 A Note on Deep Learning

If you have kept up with Artificial Intelligence on the news over the last decade, you may have heard of *Deep Learning* algorithms a number of years

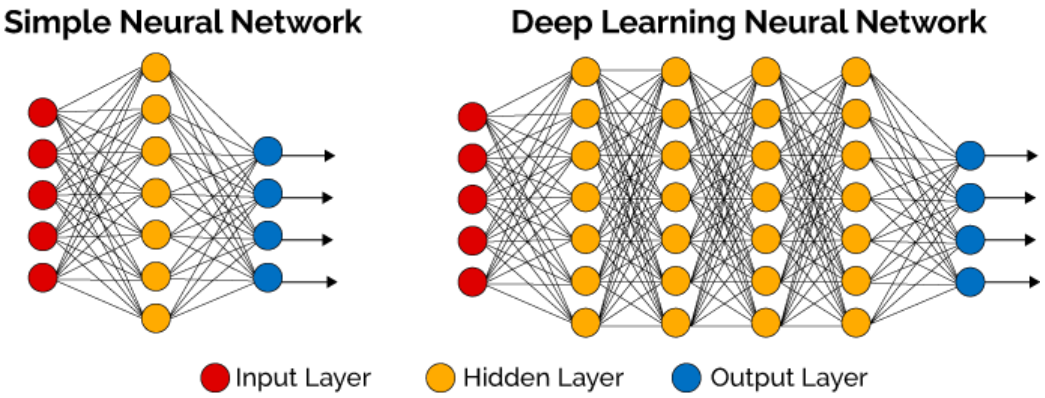

Figure 4: A contrast between a simple neural network vs a deep learning neural network. Courtesy of Vazquez (2017).

ago. Having risen in popularity early this decade, Deep Learning is a relatively new field in Machine Learning where the hidden layer is many layers deep (hence the name), with the intent to emulate the neocortex of the human brain Hof (2013). In other words, in a deep learning algorithm the hidden layer is a matrix of neurons, rather than a vector.

One of the pitfalls of deep learning algorithms is that because there are so many neurons subdivided into many hidden intermediary layers that it's almost impossible for any human to grasp the effect of any one neuron (or even a collection of neurons) on the overall output of the neural network. This is covered in more depth in Section 2.7.

Deep learning, along with *Generated Adversarial Networls* is currently at the forefront of A.I. Research; It is currently used for all kinds of tasks, from image classification tasks to virtual assistants such as *Google Now* and Amazon's *Alexa*. Vazquez (2017). The most complex deep learning algorithms can identify objects in an image almost as well as a human can.

2.5 Forward Propogation

The 'training' and 'building' process as described in section 2.1 have technical definitions: Forward Propagation and Backpropagation. *Forward Propagation* is the technical definition for the building process.

Forward propagation works as follows: for each neuron Y in a given layer, take the dot product of each neuron the in the previous layer and the synapse

associating it with Y. Equation 2 in section 2.2 models this process. This is the process that adjusts the values of the neurons of the neural network; the synapses and biases are left untouched in this stage.

2.5.1 Bias for Inactivity

For some tasks, it would be prudent to add some *bias* variable b to the forward propagation function. This bias variable adds a bias to the algorithm by setting some minimum requirement for a neuron to activate, such that a neuron only activates of the weighted sum is greater than the bias. A bias variable can be added to a forward propagation functions as follows:

$$\sum_{i=1}^{n}(\vec{Y}_{l-1} \cdot w_{ji}) - b \tag{4}$$

As discussed before, each neuron is connected to every neuron from the previous layer, with each synapse's strength is weighted by some factor between 0 and 1. As described in equation 4 above, if a bias is to be added then each neuron would contain a single bias that would shift the activation function down by the size of the bias.

2.6 Gradient Descent

In order for a neural network to learn, some form of feedback algorithm must be integrated into the training process - some method for the neural network to understand how to adjust its neurons and synapses in order to improve its accuracy each training cycle.

As as stated in section 2.1, the neurons and synapses are initialized randomly. Logically, you would conclude that the output of the neural network using the initial permutation of neuron and synapse values would lead to a meaningless output neuron. In the context of our experiment and objectives, our neural network could output anything from 0.999 to 0.453. Thus, it is crucial to figure out a method to somehow pare down the error each training cycle.

To calculate the margin of error of the network's output, we can take the square difference between the neural network's output Y_o and the expected output o. This is called the sum squared loss function. It is defined as in Equation 5.

$$E(o, Y_o) = \sum (o - Y_o)^2 \tag{5}$$

This is effective as larger discrepancies between the Y_o and o will lead to larger corrections to the synaptic weights of the algorithm and vice-versa. Typically, the formula would sum and take the average of all of the neurons in the output layer, however since in our experiment we only have one neuron, the summation evaluates to $E(o, Y_o) = (o - Y_o)^2$, so it can be disregarded for this experiment.

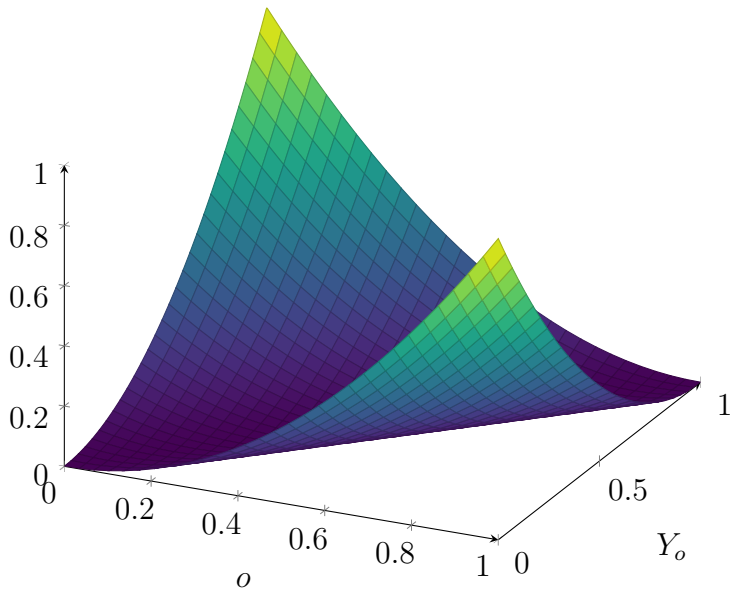

Figure 5: Graph of $E(o, Y_o)$

For instance, imagine that $o - Y_o = 0.9$. In that case, the synaptic weights that lead the neural network to a certain conclusion would be very incorrect. Squaring 0.9 gives a final value of 0.81, which is a very large change as it instructs the synaptic weights to shift towards the correct value o by 81%.

Contrast that with the case that $o - Y_o = 0.1$, In that case the neural network would be very close to being correct. Squaring 0.1 gives 0.01, which is a very small change as the synaptic weight shifts towards the correct value o by only 1%.

As stated in section 2.1, the objective of each training cycle is for the neural network algorithm to become a little more accurate in each training

cycle. Equation 5 describes the error of the output of any given training cycle. Thus, we need to find the minima of Equation 5.

To visualize finding the minima of a multivariate function, imagine trying to go as low as possible a point on the plane described by the function. The First Derivative Test suggests that for any function $f(x)$ its extrema can be found in the set $\frac{df}{dx} = 0$.

In the case of $E(o, Y_o)$ in this experiment, the variable o can only ever be one of two values: 0 or 1. This is because o represents the output of an XOR gate (as defined in section 1.2). Thus, we can treat it as a constant that represents either 0 or 1.

You can visualize this by 'mentally rotating' Figure 5 such that you are looking at it from a 'side on' perspective, such that the parabolic shape is perpendicular to your point of view- then taking the derivative of that '2-dimensional' curve.

With that in mind, we can find the minima of $E(o, Y_o)$ by taking its partial derivative with respect to Y_o - the variable that will adjust with each training cycle - and setting that to be equal to zero. This is shown in Equation 6.

$$\frac{\partial E}{\partial Y_o} = 2(Y_o - o) \tag{6}$$

Thus, by setting $\frac{\partial E}{\partial Y_o} = 0$ we can find the minima of $E(o, Y_o)$.

This technique is known as *Gradient Descent*. By knowing in which direction to alter the synapses, the neural network algorithm can improve its accuracy with each training cycle.

2.7 Backpropagation

Backpropogation is a submethod of the gradient descent algorithm, in which the weights and biases of a neural netowrk are actually adjusted. It is the step that delivers feedback into the neural network, and the process through which the neural network learns. While some may prefer the 'black box' approach to implementing the hidden layer- in which the algorithms are applied to the neural network without worrying about the small details- it is worth covering as understanding how the neural network backpropogates grants the programmer an insight into the inner workings of the algorithm, despite not fully grasping how the network interprets the values of the neurons to arrive at its conclusion as stated in Section 2.1. Note that this analysis is only

applicable in simpler neural networks. The deeper the intemediary hidden layers are, the more opaque the neural network becomes to the programmer.

As stated in section 2.3, The synaptic weights represent the strengths of the connections between some neuron Y and every neuron in the preceding layer. In other words, each Y in some layer l corresponds to some weight vector $\vec{w}$, whose components represent the strength of the link between Y and every neuron in layer $l-1$.

Another thing to keep in mind when I say that $\vec{w}$ is a vector, I mean that in the computer science interpretation of a vector, not the physics interpretation. Consider $\vec{w}$ not as an arrow in some coordinate space R^n, but as an array; or an ordered list of numbers. At first glance, you would not be entirely incorrect in thinking this distinction is meaningless in the context of this experiment. As seen in Figure 2, Any neuron will only ever have a maximum of three synaptic weights connected to it, which is perfectly visualizable in a two-dimensional or three-dimensional space. However, this approach will not hold up in larger neural networks, or even this same one with a larger intermediate layer. As Sanderson (2017c) says, it would be more easily visualizable if the magnitude of each component in $\vec{w}$ represents how sensitive $E(o, Y_o)$ is to each weight.

Equation 4 describes the factors that affect the activation of any given neuron Y in layer l:

- the bias of the neuron Y
- the weight vector $\vec{w}$, whose components represent a synaptic link between Y and each neuron in $l-1$
- the activation of each neuron in $l-1$

The neurons in $l-1$ with the highest activations have the greatest impact on the activation of Y, because as observed in Equation 2 and Equation 4, the activation of a neuron is calculated by taking the dot product of a neuron and its corresponding weight vector.

Neuropsychologist Donald Hebb once stated that "Neurons that fire together, wire together". The Hebbian theory of learning suggests that the more some process is repeated, the better the brain becomes at doing it. From his book, *The Organization of Behaviour*:

> "Let us assume that the persistence or repetition of a reverberatory activity (or "trace") tends to induce lasting cellular changes

> that add to its stability. ... When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased." - Hebb (1949)

It is by this process that backpropagation is inspired by. While it is true that more modern advancements in neuroscience have shown that artificial neural networks work far less similarly to how our brains actually work than we initially believed, it still stands as a well defined objective for backpropagation.

With that in mind, an individual weight w also functions as a physics vector; despite only having one component, it is not a scalar; its values sign indicates in which *direction* a neural network should descend $E(o, Y_o)$, and its magnitude suggests by *how much* to move it in that direction per training cycle.

This is where this concept of backwards propagation fits in; by calculating the error of the output layer, we can figure out how to adjust the weights of the previous layer. And it doesn't stop there; this method is applied recursively for each neuron in layer l with respect to any set of weights and neurons in layer $l-1$, like a wave from a ripple *propogating backwards* through the neural network. Nielsen (2015).

In the context of our research experiment, consider the activation of the output neuron $a_k^{(L)}$ (previously denoted as o) in the output layer L, such that $a_j^{(L)}$ is neuron j of layer L. Equation 5 in Section 2.6 details the error function $E(o, Y_o)$, such that o is the desired/expected output and Y_o is the output produced by the neural network algorithm.

Also, rather than associating each neuron with a weight vector $\vec{w}$, I will associate each *layer* of neurons with a *vector of vectors* - That is to say, a matrix of synapses, denoted $w_{jk}^{(L)}$, such that any weight $w_{jk}^{(L)}$. is the synaptic link between the k^{th} neuron in the layer $L-1$ and the j^{th} neuron in layer L.

To illustrate the mathematics better, I will separate the forward propogation function into three functions:

$$z_j = w_{jk}^{(L)} a_k^{(L-1)} + b_j^{(L)}$$

$$a_j^{(L)} = \sigma(z_j)$$

$$E_0(a_j^{(L)}, o) = (\sum_{j=0}^{n_l-1} a_j^{(L)} - o_j)^2$$

In other words,E_0 depends on o_j (the expected output) and $a_j^{(L)}$, which depends on z_j, which depends on the variables w_{jk}^L, $a_k^{(L-1)}$,and $b_j^{(L)}$. Keep this in mind as we proceed. Each 'level' of variables depends on the output of the previous level.

Recall again that the purpose of backpropagation is to apply the gradient descent algorithm in such a way that the neural network calculates the error of the predicted output and uses that information to adjust the synaptic weights in the system.

To put it simply, we need to find the derivative of the error function with respect to the weights. As seen in Equation x, we can use a partial derivative to take the partial derivative of $E_0(a_j^{(L)}, o$ with respect to $a_j^{(L)}$ to calculate the direction to adjust the weights. However, Equation x takes the derivative of E_0 with respect to $a_j^{(L)}$, not some element of w_{jk}.

Fortunately, as we have demonstrated earlier in the section, E_0 is a composition of $a_j^{(L)}$, which in turn is a composition of $\sigma()$ and z_j, which is a composition of $w_{jk}^{(L)}$, $a_j^{(L)}$ and $b_j^{(L)}$.

Therefore to calculate the effect any individual synaptic weight w has on the final error E_0 we can take the partial derivative of E_0 with respect to w_{jk} using the *chain rule* of differential calculus, as seen in Equation 7. The final result is presented in Equation 8.

$$\frac{\partial E_0}{\partial w_{jk}^{(L)}} = \frac{\partial E_0}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \tag{7}$$

$$\frac{\partial E_0}{\partial a_j^{(L)}} = 2(a_j^{(L)} - o)$$

$$\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = \sigma'(z_j^{(L)})$$

$$\frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} = a_k^{(L-1)}$$

$$\frac{\partial E_0}{\partial w_{jk}^{(L)}} = 2(a_j^{(L)} - o)\sigma'(z_j^{(L)})a_k^{(L-1)} \tag{8}$$

You may have noticed that $\frac{\partial E_0}{\partial a_j^{(L)}}$ is similar to Equation 6. This is because the two functions are, variable names aside, the exact same.

Another way of approaching this problem would be to consider that the change in any individual weight w has a very small effect on the overall change in the overall error E_0. Or, in other words:

$$\delta_j^{(L)} = \frac{\partial E_0}{\partial z_j^{(L)}}$$

$$\delta_j^{(L)} = \frac{\partial E_0}{\partial a_j^{(L)}} \sigma'(z_j^{(L)})$$

In most use cases of artificial neural networks, particularly classifier-type tasks, there will be multiple neurons in the output layer, in which case the average value of the function E_0 must be taken:

$$\frac{1}{n} \sum_{l=0}^{n-1} \frac{\partial E_l}{\partial w_{jk}^{(L)}}$$

This returns the average error across all of the neurons in the output layer.

Recall, however, that not only is the error function based on the synaptic weights, but on the bias variable of each neuron, too. Thus, we must also find $\frac{\partial E_0}{\partial b_j^{(L)}}$

$$\frac{\partial E_0}{\partial b_j^{(L)}} = \frac{\partial E_0}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial b_j^{(L)}}$$

The only new equation this differentiation results in is $\frac{\partial z_j^{(L)}}{\partial b_j^{(L)}}$, defined as thus:

$$\frac{\partial z_j^{(L)}}{\partial b_j^{(L)}} = 1$$

This is very convenient, because the rate of change of $\frac{\partial z_j^{(L)}}{\partial b_j^{(L)}}$ is a constant 1. Which implies that the rate of change of the error E_0 with respect to the bias is directly proportional to the output of some infinitesimal error produced

by the neural network. The function E_0 is simply $\sigma(z)$, so the rate of change of b does not change relative to z. This is illustrated in Equation 9.

$$\frac{\partial E_0}{\partial b_j^{(L)}} = \delta_j^{(L)}$$
$$\frac{\partial E}{\partial b} = \delta \tag{9}$$

This generalized process can be applied not only to the output layer but to every single layer in the neural network algorithm. This is where the concept of propagating backwards comes from; this process is applied recursively to each neuron in each layer of the neural network, save for the input layer.

There's a small catch, however; bear in mind that a change in some neuron in layer $L-1$ affects not only a single neuron $a_j^{(L)}$ in layer L, but every neuron $a_j^{(L)}$, $a_{j+1}^{(L)}$, $a_{j+2}^{(L)}$, etc. Therefore, in the case $\frac{\partial E_0}{\partial a_k^{(L-1)}}$, the neurons affected must be added up:

$$\frac{\partial E_0}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_L-1} \frac{\partial E_0}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}}$$

The result of all these calculations, and the result we are looking for, is the gradient vector of the function E_0, composed of the partial derivatives of E_0 with respect to $w_{jk}^{(L)}$ and of E_0 with respect to $b_j^{(L)}$. This is shown in Equation 10. You can think of it as a vector that 'points' towards the direction the neural network needs to adjust towards.

$$-\nabla E = \begin{bmatrix} \frac{\partial E}{\partial w_{11}^{(1)}} \\ \\ \frac{\partial E}{\partial b_1^{(1)}} \\ \\ \vdots \\ \\ \frac{\partial E}{\partial w_{jk}^{(L)}} \\ \\ \frac{\partial E}{\partial b_j^{(L)}} \\ \\ \end{bmatrix} \tag{10}$$

2.7.1 Stochastic Gradient Descent

Admittedly, a pure gradient descent algorithm is rarely used in practical applications of artificial neural networks. More commonly, a slightly different method called *Stochastic Gradient Descent* is used.

A gradient descent algorithm as defined above, where each training sample data is passed through every single neuron and synapse is computationally expensive. Very computationally expensive. So instead a stochastic gradient descent algorithm implements backpropagation in mini-batches of training data, which is initially less accurate but far less computationally expensive.

I will not be using any form of stochastic gradient descent as I am using the simplest form of an artificial neural network, and predicting the output of an XOR gate is far too simple to of a task to be able to implement stochastic gradient descent anyway.

3 Research Methodology

3.1 Why Use Python 3.7.3?

There are several reasons to use Python. It is the industry standard for any type of machine learning or artificial neural network algorithm.

Because Python has been around for so long and continues to remain popular, it has a large ecosystem of libraries and APIs that make it a convenient choice for programming a complex structure such as an artificial neural network. APIs written in the C programming language, such as numpy, scipy, and pyplot, handle all the backend mathematics faster than any hand-coded program in any other high-level programming language.

Furthermore, Python is a very high-level and easily readable language. When dealing with structures as complex as artificial neural networks, it helps if the programmer doesn't need to worry about lower-level operations (such as memory management and pointer arithmetic).

I could have used Java, but it's a very verbose language- and with an already very complicated data structure, the logical and mathematical process. Additionally, most Java libraries are written in Java (and even those that aren't are wrapped in a Java interface), so it would be more computationally expensive with no added benefit to make up for it.

3.2 Numpy.py

Numpy is a Python package most often used by data scientists, engineers, and mathematicians in favour of Matlab. It has a unique advantage in that it is written in C with a Python frontend, making it both fast and easy to use and read.

It also has a lot of mathematically complicated functions that we will need; such as matrix transposes, dot multiplication, as well as native vector and matrix data types designed to be used with the aforementioned functions.

3.3 Why Use a Vanilla Artificial Neural Network?

As stated in Section 1.3, I chose to use a simple artificial neural network algorithm with no added frills. While it is true that, theoretically, so long as I use the same algorithms to create the neural network the change in efficiency of an additional neuron would be consistent, I wanted to use the simplest possible artificial neural network so as to emphasize and keep the focus on the objective of the experiment.

I am also concerned that a different, more optimized set of algorithms might alter the results of the research experiment. A different error function, for instance, would completely change the formulas derived in Section 2.7. This may have an effect on how efficient the neural network is.

3.4 Why Run So Few Training Cycles?

Given a sufficient number of training cycles, all artificial neural network algorithms have the potential to be highly accurate. As stated in Section 1.2, the aim of this experiment is to calculate and analyze the effect of a marginal node in the intermediary layer of the artificial neural network.

If the number of training cycles is high enough, then no matter the number of neurons the network will make very accurate predictions. In which case, the output of the neural network will be useless for the purposes of our experiment.

At the same time, we have to be careful to not choose *too* few training cycles, lest the neural network output seemingly arbitrary numbers. We'll have to just keep slowly increasing the number of training cycles - like one would tune the frequency of an FM radio- until we get workable data.

Ultimately, I settled for 250 training cycles.

3.5 Why Disregard the Number of Training Cycles With Regards to Efficiency?

We can derive a formula to calculate the predicted efficiency of the artificial neural network. We know that the efficiency of a neural network is generally directly proportional to the number of neurons in each intermediary layer. We also know that (because we are trying to maximize the change in accuracy per training cycle) that the efficiency of the network is inversely proportional to the number of training cycles. The result of this derivation is described in Equation 11, such that L is a label denoting the current layer, Y is the number of neurons in that layer and t is the total number of times the neural network undergoes a training cycle.

$$\eta(Y,t) = \frac{Y^{(L)}}{t} \tag{11}$$

The partial derivatives of Equation 11 with respect to Y and t are described in Equations 12 and 13. They show that for any value t, $\frac{\partial\eta(Y,t)}{\partial Y}$ is more sensitive to change than $\frac{\partial\eta(Y,t)}{\partial t}$. We know this because for all t such that $t > 1$, $\frac{1}{t^2}$ is smaller than $\frac{1}{t}$. Therefore no matter what variables are adjusted, increasing the number of neurons is exponentially more efficient than increasing the number of training cycles.

$$\frac{\partial\eta(Y,t)}{\partial Y} = \frac{1}{t} \tag{12}$$

$$\frac{\partial\eta(Y,t)}{\partial t} = -\frac{Y}{t^2} \tag{13}$$

4 Research Results

The figures in this section are a graphical representation of the output of DataCollector.py in Section 6.2. Each line represents one of three sets of training cycles - each with a freshly instantiated artificial neural network. In each case the neural networks were trained 250 times. Figures 6, 7, 8, 9 represent each potential inputs/output pair (or *cases*) of a logical XOR gate, as detailed in Figure 1, in section 1.2.

4.1 Case 1

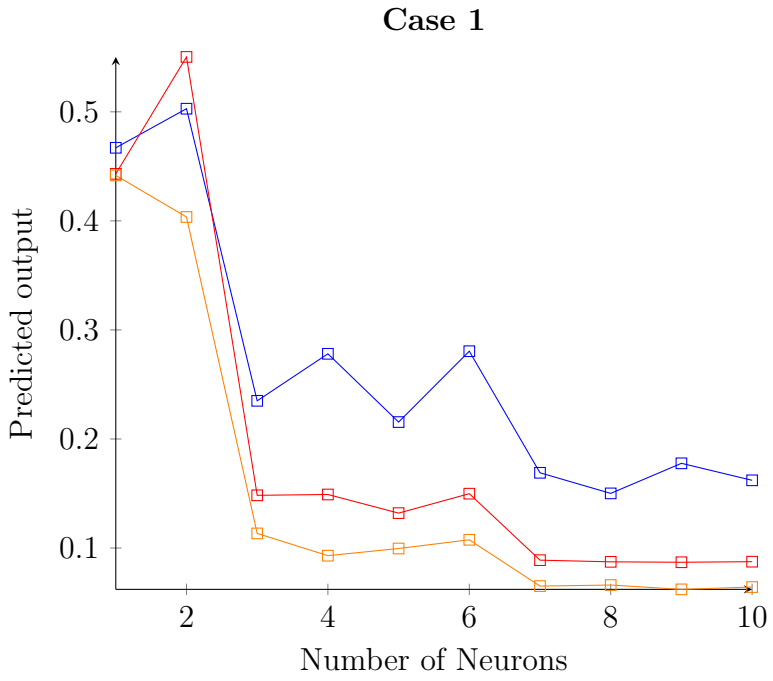

Figure 6: Predicted outputs of Case 1 after 250 training cycles

4.2 Case 2

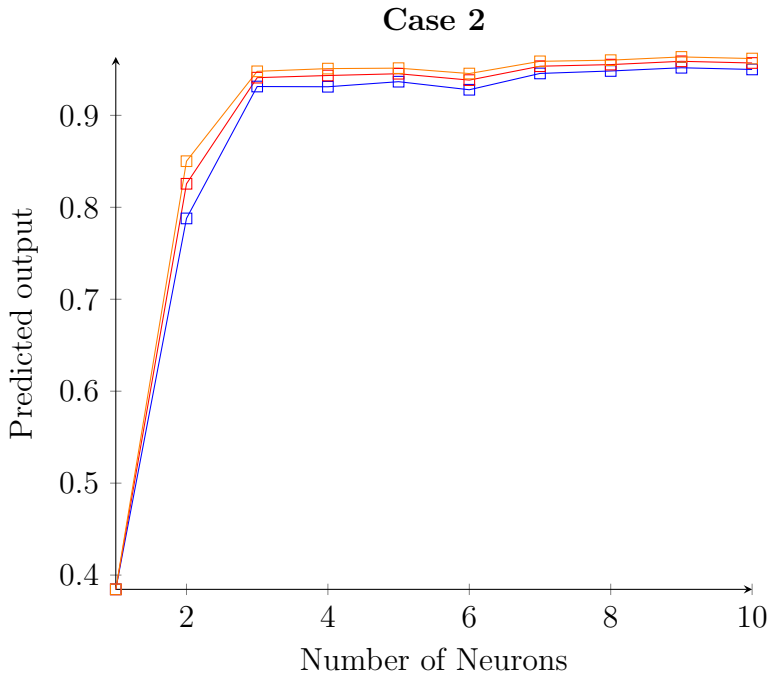

Figure 7: Predicted outputs of Case 2 after 250 training cycles

4.3 Case 3

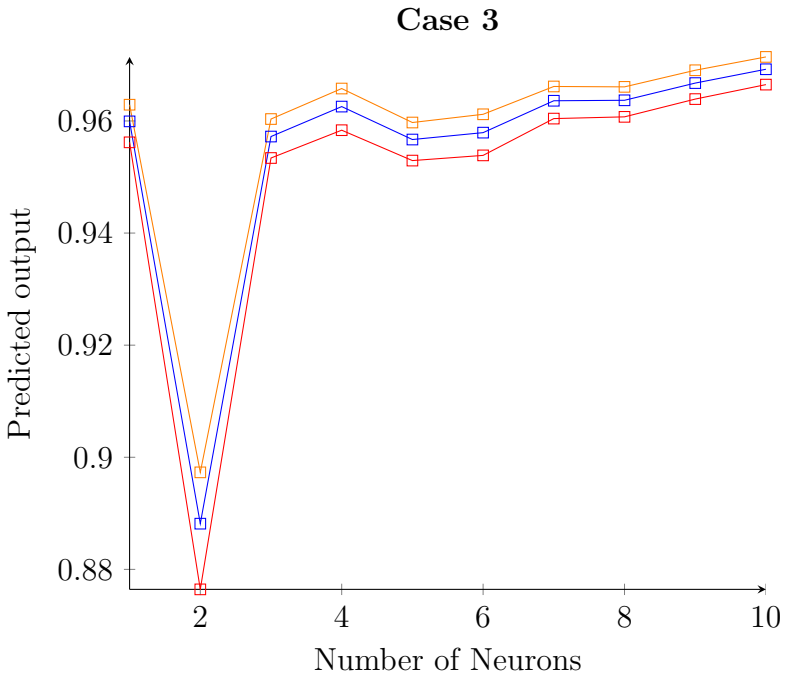

Figure 8: Predicted outputs of Case 3 after 250 training cycles

4.4 Case 4

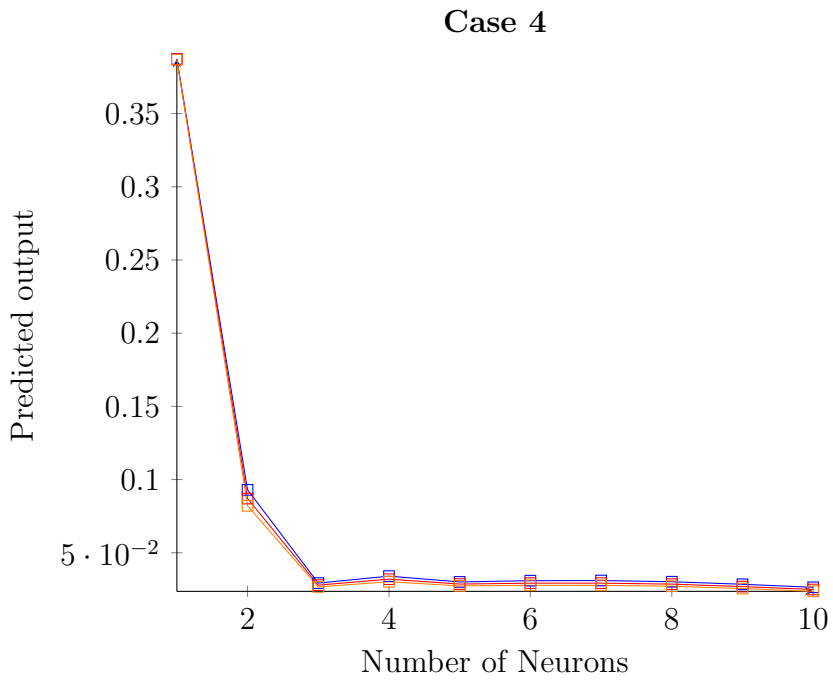

Figure 9: Predicted outputs of Case 4 after 250 training cycles

5 Conclusions and Limitations of This Research

5.1 Conclusions

As can be seen in figures 6, 7, and 9, at $n = 1$ (such that n is the number of neurons) the neural network produces complete gibberish. It never varies much beyond $Y \approx 0.5$. There is a sharp increase in certainty at $n = 2$ - interestingly, this is also the number of neurons in the input layer. This is not the case in Figure 6, which in two out of three attempts sees a *decrease* in efficiency going from $n = 1$ to $n = 2$. Case 1 is unique in that all input neurons have a value of 0. This may mean that in terms of Big-O Efficiency Notation, Case 1 is the worst possible case. This unique inconsistency in Figure 6 may also suggest that the output of the neural network at $n = 2$

is also arbitrary as in the case of $n = 1$, because the input layer in Case 1 is $[0, 0]$ - so the algorithm can't immediately correct itself, since any value x multiplied by 0 is equal to 0. In all other cases save for Case 3 (which will be explained in more detail further down this section), All neural networks experience a ubiquitous sharp increase in efficiency - indeed, the network only starts to make accurate predictions then - at $n = 2$.

This is likely because the input layer is also two neurons long. At $n = 2$, the number of synaptic weights between the input layer and the intermediary layer is 4 - the same number of the set of possible input/output pairs of an XOR gate. Thus, we can conclude that at $n = 2$, each synaptic link between neurons represents the likelihood of output given a certain set of inputs. This is the minimum number of synaptic weights required for the artificial neural network to function. At $n > 2$, the number of synaptic weights increases exponentially, allowing for more subtle decision-making. This means that the neural network can make more accurate predictions each training cycle.

At $n = 3$, the neural network algorithm again experiences a sharp increase in effeciency, albeit not as sharp of an increase as that at $n = 2$. As shown in Figure 6, The neural network that was least accurate in $n = 2$ - the red one - experienced that greatest correction in accuracy in $n = 3$. This is the Gradient Descent Algorithm (discussed in Section 2.6) working precisely as intended.

From there, the prediction lines generally slope towards their intended targets, though the algorithm does tend to make errors in the range $4 \leq n \leq 6$. This may validate my hypothesis in Section 1.2, where I suggested the efficiency of the neural network may be modelled by $O(nlog(n))$.

Figure 8 is a very bizarre anomaly. At $n = 1$ the neural network's predicted output is very close to the actual output of a logical XOR gate. Then at $n = 2$, the neural network is less accurate than it was at $n = 1$. Furthermore, upon close examination you will notice that the gradient descent algorithm still applies- the further a prediction line is from $Y = 1$ at $n = 1$, the greater the correction was to the algorithm at $n = 2$ - except *it corrected in the wrong direction.* Perhaps the gradient vector had somehow been scaled by a negative factor?

I stated in Section 2.1 that the human programmer can only glimpse how any specific artificial neural network 'thinks', via analyzing the backpropagation algorithm. At higher number of neurons, the artificial neural network has no difficulty correctly guessing the outputs of all four cases of an XOR gate; but this is an intriguing glimpse at its thought process regardless.

5.2 Limitations

Using the output of an XOR gate as a training example serves the experiment well in that it is simple . However it carries with it some limitations.

Since an XOR Gate only has two inputs, and n is a set of natural numbers, We can only monitor a neural networks behavior when the intermediary layer is smaller than the input layer for one example: $n = 1$. While it is true that (as explored in Section) a neural network only begins to make accurate predictions when the intermediary layer is greater than or equal to the input layer, It would have been beneficial to view if there was a trend in the prediction line if n was smaller than the input layer for several n.

6 Appendix

6.1 NeuralNetwork.py

```
import numpy as np
import os
class Neural_Network:

    def __init__(self, hidden_layer_size):
        # Layers
        self.input_layer_size = 2
        self.hidden_layer_size = hidden_layer_size + 1
        self.output_layer_size = 1

        # Weights
        self.weights_input_to_hidden = np.random.randn(self.
    input_layer_size, self.hidden_layer_size)
        self.weights_hidden_to_output = np.random.randn(self.
    hidden_layer_size, self.output_layer_size)

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def d_sigmoid(self, z):
        return z * (1 - z)

    def feed_forward(self, inputs):
        self.z_L_minus_one = np.dot(inputs, self.
    weights_input_to_hidden)
        self.a_L_minus_one = self.sigmoid(self.z_L_minus_one)
```

```
24
25          self.z_L_one = np.dot(self.a_L_minus_one, self.
    weights_hidden_to_output)
26          self.a_L_one = self.sigmoid(self.z_L_one)
27
28          return self.a_L_one
29
30      def backpropogate(self, inputs, expected_output,
    predicted_output):
31          self.output_error = (expected_output - predicted_output)
32          self.output_error_delta = self.output_error * self.
    d_sigmoid(predicted_output)
33
34          self.weights_error = self.output_error_delta.dot(self.
    weights_hidden_to_output.T)
35          self.weights_error_delta = self.weights_error * self.
    d_sigmoid(self.a_L_minus_one)
36
37          self.weights_input_to_hidden += inputs.T.dot(self.
    weights_error_delta)
38          self.weights_hidden_to_output += self.a_L_minus_one.T.
    dot(self.output_error_delta)
39
40      def train(self, inputs, expected_output):
41          predicted_output = self.feed_forward(expected_output)
42          self.backpropogate(inputs, expected_output,
    predicted_output)
```

6.2 DataCollector.py

```
import NeuralNetwork as nn
import numpy as np
import os

MAX_NUMBER_OF_NEURONS = 10                  +1
NUMBER_OF_TRAINING_CYCLES = 250             +1
NUMBER_OF_CASES = 4                         +1
NUMBER_OF_ATTEMPTS = 3                      +1

XOR_gate_inputs = np.array( ([0, 0], [0, 1], [1, 0], [1, 1]), dtype=float)
XOR_gate_outputs = np.array( ([0], [1], [1], [0]), dtype=float)

class DataCollector:

    def __init__(self):
        print("initializing Data Collector\n ------------------------------")

        print("creating network_array")
        self.network_array = []
        for i in range (0, MAX_NUMBER_OF_NEURONS - 1):
            self.network_array.append(nn.Neural_Network(i + 1))

        self.case = np.zeros([NUMBER_OF_CASES, NUMBER_OF_ATTEMPTS]) # 2d array of [case][attempt number], storing the final result after x training cycles


    def attempt_for_all_neuron_sets(self, case_number, attempt_number):
        for number_of_neurons in range(0, MAX_NUMBER_OF_NEURONS - 1):
            print("\t\tNumber of Neurons: ", number_of_neurons + 1)
            for cycle in range(1, NUMBER_OF_TRAINING_CYCLES):
                self.network_array[number_of_neurons].train( XOR_gate_inputs, XOR_gate_outputs)
            self.case[case_number, attempt_number] = self.network_array[number_of_neurons].feed_forward(XOR_gate_inputs)[case_number]
            print("\t\t Final Value: ",self.case[case_number, attempt_number])
```

```
33
34
35      def add_attempt(self, case_number):
36          for attempt_number in range (0, NUMBER_OF_ATTEMPTS - 1):
37              print("\tattempt number ", attempt_number + 1)
38              self.attempt_for_all_neuron_sets(case_number,
    attempt_number)
39
40
41      def cycle_through_cases(self):
42          for case_number in range(0, NUMBER_OF_CASES - 1):
43              print("Adding attempts for case ", case_number+1)
44              self.add_attempt(case_number)
45
46  dc = DataCollector()
47  dc.cycle_through_cases()
```

7 References

Grey, CGP (2017). *How Machines Learn*. Youtube. URL: https://www.youtube.com/watch?v=R9OHn5ZF4Uo.

Hebb, Donald (1949). *The Organization of Behaviour*. Psychology Press.

Hof, Robert D. (2013). "Deep Learning: With massive amounts of computational power, machines can now recognize objects and translate speech in real time. Artificial intelligence is finally getting smart." In: *MIT Technology Review*. URL: https://www.technologyreview.com/s/513696/deep-learning/.

Nielsen, Michael A. (2015). *Neural Networks and Deep learning*. Determination Press.

Sanderson, Grant (2017a). *But what *is* a Neural Network? — Deep learning, chapter 1*. 3Blue1Brown. URL: https://www.youtube.com/watch?v=aircAruvnKk.

— (2017b). *Gradient descent, how neural networks learn — Deep learning, chapter 2*. 3Blue1Brown. URL: https://www.youtube.com/watch?v=IHZwWFHWa-w.

— (2017c). *What is backpropagation really doing? — Deep learning, chapter 3*. 3Blue1Brown. URL: https://www.youtube.com/watch?v=Ilg3gGewQ5U.

Shamdasani, Samay (2017). "Build a Neural Network". In: *enlight*. URL: https://enlight.nyc/projects/neural-network/.

Sharma, Avisnash (2017). "Understanding Activation Functions in Neural Networks". In: *The Theory of Everything*. URL: https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0.

Vazquez, Fabio (2017). "Deep Learning made easy with Deep Cognition". In: *BecomingHuman.ai*. URL: https://becominghuman.ai/deep-learning-made-easy-with-deep-cognition-403fbe445351.