



Zeeshan Abid

BSc(Hons) Computer Games (Software Development)

Matric Number: S1314747

## Honours Project Final Report

Module Leader: Brian Shields

**The Most Effective Way to Train a Multilayer Feedforward Neural Network: Comparing  
Back Propagation, Particle Swarm Optimisation, and Genetic Algorithm**

Project Supervisor: Dr David Moffat

2<sup>nd</sup> Marker: David Farrell

*“Except where explicitly stated, all work in this report, including the appendices, is my own original work and has not been submitted elsewhere in fulfilment of the requirement of this or any other award”*

Signed: \_\_\_\_\_

Date: \_\_\_\_\_

## Contents

1. Introduction	4
1.1. Project background	4
Neural Networks	4
Back Propagation	5
Particle Swarm Optimisation	5
Genetic Algorithm	6
1.2. Research Question	6
1.3. Project Outline	7
1.4. Project Objectives	7
[S1] Research on Neural Networks	7
[S2] Research on Each Technique For Training Neural Network	8
[S3] Planning the Experiment	8
2. Literature Review	9
2.1. Neurons & Synapses	9
Bias	10
Activation Functions	10
Multilayer Feed-Forward Neural Network	11
2.2. Training a neural network	12
Supervised vs Unsupervised	12
Training Error	12
Model Overfitting	12
Deep Learning	12
Back propagation	13
Particle Swarm Optimisation	15
Genetic Algorithm	18
3. Methods	20
3.1. Methodologies & the Experiment	20
Resources	21
3.2. Inner workings of the project	21
Free Parameters	22
Data Re-ordering	22
Data Normalisation	23
Activation Function	23

Training Error	23
3.3. Hypothesis	23
4. Implementation & Testing	24
4.1. Class Diagram	24
4.2. Project Life Cycle	25
4.3. Implementation	26
Activation Functions	27
Neuron & Synapse	29
Multi-layered Feed-Forward Neural Network	31
Back Propagation	36
Particle Swarm Optimisation	41
Genetic Algorithm	46
Training Data	51
4.4. Testing	52
Activation Functions	52
Neurons, Synapsis & MFNN	54
Back Propagation, Particle Swarm Optimisation & Genetic Algorithm	59
5. The Experiment	63
5.1. Preparation	63
5.3. Results, Analysis & Discussion	63
Iris Data Set	64
Breast Cancer	68
Wine	72
Car Evaluation	77
6. Future Work	81
7. Conclusion	81
8. References	82
9. Appendix	85
The Code	85
Activation Header File	85
Activation CPP File	86
Core Functions Header File	89
MFNN Header File	90
MFNN CPP File	91

Base Network Header File	101
Neuron Header File	102
Neuron CPP File	103
Synapse Header File	105
Synapse CPP File	106
GA header File	106
GA CPP File	107
PSO Header File	111
PSO CPP File	112
Common Header File	116
Main CPP File	117
Experiment Results & Outputs	118
BP Breast Cancer Averages	118
BP Car Evaluation Averages	119
BP Iris Averages	120
BP Wine Averages	122
GA Breast Cancer Averages	123
GA Car Evaluation Averages	124
GA Iris Averages	125
GA Wine Averages	126
PSO Breast Cancer Averages	127
PSO Car Evaluation Averages	128
PSO Iris Averages	130
PSO Wine Averages	131
UCI Machine Learning Data	132
Iris Data Set	132
Wine Data Set	133
Car Evaluation Data Set	134
Breast Cancer Data Set	136

## **1. Introduction**

The human brain is incredibly complex, its ability to solve difficult problems in day to day life is astounding. The way humans make sense of their surroundings, observe, and put themselves in another's shoes is remarkable. From birth, there are no instructions given to a child but his/her curiosity to learn and explore which leads them to understand the world. Does the child know how to walk from birth or does he/she mimic others and learn how to walk? Despite years of research and studies conducted on the human brain we still fail to replicate the ingenious design of human brain; resorting to cheap tricks known as AI. These techniques are getting impressive by the year, but yet they still follow instructions programmed into them, unable to think freely.

After the famous victory over world chess champion Garry Kasparov by a computer known as Deep Blue people thought computer intelligence would skyrocket and surpass humans quickly. However, in actuality, the human mind uses past experiences and strategies, whereas deep blue AI goes through all possible outcomes yet to come to determine the best path in a tree like structure. Would that be considered as an intelligent or menial task being performed by a computer? Another example is the Jeopardy champion named Watson, an AI computer created by IBM. Watson took a different approach than deep blue since winning Jeopardy requires more than a search tree through all the possibilities. Watson was built from neural networks, and Watson was not only programmed but trained with a training data regularly improving its skills each day with new information.

### **1.1. Project background**

#### **Neural Networks**

The neural networks are a computational model, designed to mimic synapses and neurons in the human brain. Created in 1943 it quickly fell out of favour for alternative methods as it was not practical in doing extensive computations (McCulloch, Pitts 1943). However, with the advancements in electronics and computers, the neural networks have risen again. Instead of doing the calculation on paper, computers can take advantage of the technique with their tremendous speed.

Neural networks by themselves are not able to do much. Neural networks require training. Otherwise, it is an empty shell without an end goal. In its simplest form, neural networks work off inputs and outputs. Giving them inputs and expected outputs the neural networks will then reconfigure themselves slowly to match those expected outputs (Maind, Wankar 2014). Neural networks are extraordinary having applications across a wide range of fields. Today neural networks are used for stock market prediction, breast cancer detection, image recognition, voice recognition and much more. There are several techniques brought forward to train a neural network. Like the layout of the keyboard is dominated by "qwerty" format even though it might not be the most efficient way of typing. Back-propagation technique dominates the world of neural networks (Werbos 1990).

## **Back Propagation**

BP (Back-Propagation), the most commonly used technique for training neural networks is usually coupled with an optimization method such as gradient descent. BP starts off from the end point of neural networks where the outputs are computed. After comparing these outputs to the expected results, BP goes through a neural network backward changing weights slightly to produce the lowest error possible (Rumelhart, Williams 2013). BP is the only technique that is an algorithm, not a metaheuristic that is being compared in this paper. Having BP around for decades in the world of neural network has brought forth various enhancements/optimizations that can be applied to BP, giving superior results in training neural networks (Saduf, Wani 2014). The four top BP optimizations currently being used are as follows.

### **Momentum Strategy:**

One of the issues BP has is that it relies on learning rate. If the learning rate is made too small, a neural network will need to take more steps in changing weights by minuscule amounts. However, if the learning rate is increased too much than, a neural network can become unstable rapidly evolving weights unable to get any desired results. This problem can be resolved by introducing gradient descent. A technique that increases/decreases the learning rate depending on how large the gradient of the curve is.

### **Using Error Saturation Prevention Function:**

In the neural networks, activation function "logistic sigmoid" can cause slow learning using BP if the output of a unit is near 0 or 1. This term is called "error saturation." There have been a few proposals to change the sigmoid function to minimize this error saturation.

### **Using Weights Initialisation Method:**

BP is sensitive to the initial weights since it has to propagate the error backward using the result it gets with initial weights. Selecting initial weights close to the actual minimum can massively reduce the training time. A wide range of techniques has been brought forth to solve this conundrum.

### **Adjusting the Steepness (slope) of Sigmoid Function:**

The surface of the sigmoid function consists of flat and extremely steep gradient due to the nature of the sigmoid function. This can cause the outputs of a neuron to get stuck into a high error level area while weights are being adjusted by a small amount. This issue can be resolved by reducing the steepness of sigmoid function.

## **Particle Swarm Optimisation**

Some pioneers in the field have described the discovery of PSO (Particle Swarm Optimisation) as an observation of social behaviour, specifically the study of flocking patterns of birds & fish (Kennedy, Eberhart 1995). The ability to communicate between a massive school of fish or flocks of bird is perplexing. Ask any fighter jet, and they will explain it is no easy task to fly in such a strict manner. Having a single mind and never crashing into one another. A suggestion that global information is shared with each member during the

search for food (Wilson 1975). A global variable accessible by the entire herd is the backbone of PSO and gives an enormous evolutionary advantage. Groups of birds and Fish adjust themselves to avoid predators while seeking for food and mates. At times, environmental changes can also be considered, the change in temperature could lead the herd a different path.

Transposing PSO to neural networks a simulation of flocks of birds or school of fish can be performed to find the lowest error in the neural network. PSO is no algorithm, but a metaheuristic, a set of paradigms that have to be implemented and each implementation can vary. PSO generates a handful of particles those particles each with velocity, position, best position, error, and best error. There are also global variables accessible by the entire swarm best global position and best global error. Every member applies a basic formula on how it should behave and converge onto the best solution (McCaffrey 2015).

### **Genetic Algorithm**

Genetic Algorithm is particularly appealing to its imitation of natural selection happening in everyday life. The genetic algorithm also falls under the category of metaheuristics. There is no fixed algorithm for genetic algorithm and each implementation by a different person can produce different results (Forrest 1993). An initial pool of agents is created, whereby each agent represents a potential solution. These agents in plain language are nothing more than an array of binary data. Each agent is then evaluated and given a score such as their error. This score can then be determined to eliminate the weak agents and replacing them with copies of high scoring agents. These new copies of agents then have genetic operators applied to them such as mutation and crossover (Forrest, Javornik, Perelson, Smith 1993).

Mutation is a process where an agent's array of binary data is randomly changed.

Meanwhile, a crossover is taking two separate agents and combining them. Using two different agents to create a new agent by splitting the data. Resulting in an offspring like effect, imitating mating (Forrest 1993).

One can easily go down the rabbit hole where they do not consider the possibility; there could even be a different technique to train neural network other than BP. Dr McCaffrey points out a major problem with the current state of neural networks. Blindly following BP technique to train neural network can be limited. However, finding a reputable implementation of a neural network using anything other than BP can be a daunting task (McCaffrey 2013). New Advancements are made by thinking creatively and going through unpopular paths. Raising an important question, why are other techniques not being used for training neural networks? Is BP indeed superior? If so there is a huge shortage of evidence supporting that case.

### **1.2. Research Question**

Comparing BP (Back Propagation), PSO (Particle Swarm Optimisation), and GA (Genetic Algorithm) to determine what the most efficient way to train a multilayer feedforward neural network is?

### 1.3. Project Outline

This project has an aim of testing each of the techniques (BP, PSO, and GA) on how efficient each of them is at training a multilayer feedforward neural network(MFNN). Expanding on the research already conducted showing different optimizations for BP (Saduf, Wani 2014).

To find the most effective method to train neural networks we first need to understand how neural networks function, More specifically MFNN. Explaining the workings of a MFNN is the utmost of importance regarding this paper, considering without the all in depth explanation on neural networks the conclusion could be inconclusive.

Not only the neural network itself but the techniques used to train will need extensive research and explanation. Starting with BP and inner workings of it, how can it be used to train the neural network. Then expanding on it by introducing how optimization can affect BP, improving results. The next step is to take a look at PSO. PSO being a metaheuristic will not only need to be explained how it works but also how it will be implemented into the MFNN. Since each implementation of PSO can vary depending on the person implementing PSO. Finally, GA is the last metaheuristic that will be described on how it functions. How GA is tied to the natural selection happening in daily life. Expanding further on it by going through genetic operators that will be applied to the metaheuristic. This is, useless unless it is tied to the subject of MFNN and how each genetic operator will be made to enhance the training of a neural network.

Moreover, using a '*develop and test*' methodology a neural network library will be set up and implementation of each technique will be present. These techniques will then be compared to their efficiency to converge on the correct solution, the time required for an optimal answer and processing time took per epoch. Once the data for each technique is collect it will be formatted and further analysed.

The justification for develop and test methodology is seeing how neural networks can be transferred from theoretical paper to being implemented in an application. Furthermore, adding BP, PSO, and GA on top, improving my skills for implementing an AI architecture in the future.

### 1.4. Project Objectives

#### [S1] Research on Neural Networks

This project starts off by conducting extensive research on the topic neural networks. The research on neural networks will be the foundation, building upon the entire project. Just like any structure without the underlying foundation, the project will fall.

Neural network is not a new concept by any stretch of the imagination. Considering that there is no shortage of papers released on neural networks. Including a dedicated journal on neural networks. One of the most important research paper titled "A Logical Calculus of the Ideas Immanent in Nervous Activity" is a must-read in the field of neural networks.

Throughout the year's many advancements have been made in neural networks.

Researching on those advancements, understanding how they work can improve the results of this project.



### **[S2] Research on Each Technique For Training Neural Network**

Following the analogy of neural network being the foundation of this project. Each technique is the next stage, standing on top of neural networks. It is just as necessary for the success of this project that research is done not only on the neural network, but the techniques used to train a neural network as well. In fact, doing research on neural networks highlights the fact that BP is by default included in recent papers alongside neural network (Schmidhuber 2015). Meanwhile scavenging a paper using PSO or GA with a neural network can prove to be difficult. However, this is not a sign to neglecting them from the research. PSO and GA are just as important for the outcome of this project as BP.

PSO and GA are very different from BP since they are not an algorithm but metaheuristics. While BP has mathematical equations and steps set in stone PSO and GA are a series of paradigms that have to be implemented and each implementation can differ. Again, showing the importance of researching into PSO and GA. At the end of this project producing fair results requires a reputable implementation of all three techniques. With BP being used by default in most papers it becomes even more important to research into PSO and GA.

### **[S3] Planning the Experiment**

Before starting the practical side of the project, some planning is necessary. Just as we discussed, implementation of PSO and GA can differ. In deciding what approach will be optimal, planning is mandatory. Moreover, to train a neural network and conduct the experiment several data sets are required. Choosing a broad range of data sets that challenge BP, PSO, GA, and the neural network itself to the fullest are essential. Not only that but how the MFNN will be structured. The number of hidden layers that will be present to how many neurons will be needed for a particular data set.

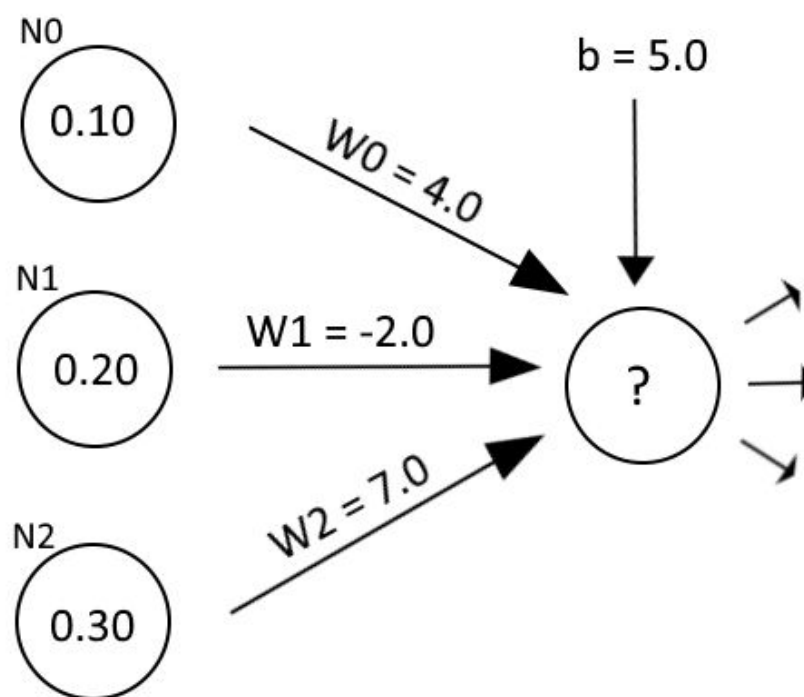
## 2. Literature Review

Making sense of a problem by using large clusters of neurons connected by synapses is a daily routine for our biological brain. Neural networks are an attempt to harness this power. To understand how neural networks or our natural brain function, we first must take a look at the ins and outer workings of a neuron and a synapse.

### 2.1. Neurons & Synapses

In a neural network, each neuron unit (also known as perceptron) has a value. Which can be computed by the neuron, or if it is an input neuron, the value will be provided by some input device. How does a neuron compute its value? A neuron unit will compute its value by using synapses that are connected to the neuron.

Synapses being unidirectional, only the synapses going to the neuron will be taken into account. Synapses provide the value to the neuron they are connected to by multiplying a weight by the value of the neuron these synapses are linked from. All the values provided by different synapses are then added. Followed by an activation function applied before calculating the final neuron value (McCulloch, Pitts 1943).



STEP 1: Multiply and add previous node values and weights.

$$\text{VALUE} = (W0 * N0) + (W1 * N1) + (W2 * N2)$$

STEP 2: Add Bias

$$\text{VALUE} = \text{VALUE} + b$$

STEP 3: Apply Activation Function

$$? = \text{Activation Function}(\text{VALUE})$$

## Bias

The diagram above shows how the value of a unit can be calculated. Take note of the bias, a constant that is added to the result. Most literature treats the bias as a synapse connected to a dummy node which is always 1.0 (McCaffrey 2013). That said in our example if we were to follow the literature the bias would be connected to another circle, with the circle having a value of 1.0. Then we skip the “STEP 2” but end up with the same result. However, in neural networks having the bias as a constant number added on at the end is easier to implement in practice (McCaffrey 2013).

## Activation Functions

Finally, the secret sauce of neural networks the activation function is applied in the last step. Without the activation function, it is just simple algebra. Essential characteristics of the activation function are providing a smooth transition as the input values change. For example, small shifts in the input produce little changes in output. Several different types of activation functions can be used in a neural network. However, a random selection of activation function is not advised as it could lead to a un functioning neural network (Mhaskar, Micchelli 1993). Moreover, each activation function outputs value in its range. For example, where logistic sigmoid will output between 0 and 1, hyperbolic tangent will output between -1 and 1. That said choosing the correct activation function for what is required for the neural network is crucial. Most of the activation functions have a basic 'S' shape shown in the below diagram. The four most common activation functions are: (McCaffrey 2013)

Logistic Sigmoid:

Output between [0, 1]

$$y = \frac{1}{1 + e^{-x}}$$

Hyperbolic Tangent:

Output between [-1, 1]

$$y = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Heaviside Step:

Output is either 0 or 1

$$y = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

Softmax:

Outputs between [0, 1] and all nodes value sums to 1.0

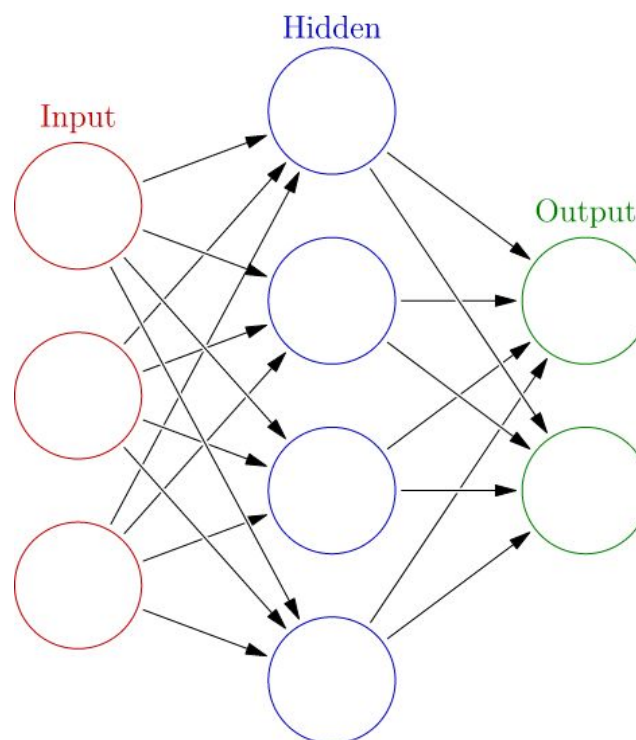
$$y = \frac{(e^{-xm})}{\sum_{n=0}^L (e^{-xm})}$$

Where m is maximum x value and L is total amount of x values.

### **Multilayer Feed-Forward Neural Network**

Neural networks are structures that can be trained to map any curve theoretically. Although there are many different variations of neural networks. Structured in various ways each having their advantages and disadvantages, this paper will focus on multilayer feedforward neural networks (MFNN). MFNN that are unidirectional work in layers, from left to right (Tang, Salakhutdinov 2013). The first layer being input layer and the far right layer being the output layer. The input layer is used to pass values to the neural network and output to extract the values. Layers laying in between input and output layers are known as hidden layers. There will be no restrictions on how many hidden layers are present as well as how many nodes are required per hidden layer (Aizenberg, Moraga 2007).

The neural networks are a very large topic having various different structures. MFNN will be used in the project because they are easy to work with and this project can be the ground work for a more complex neural network for future if someone decides to follow up.



## **2.2. Training a neural network**

### **Supervised vs Unsupervised**

The main meat of the project, training the neural network. After structuring the neural network, weights are chosen at random. Then the training for the neural network can begin. Currently, a neural network can be trained in two ways supervised or unsupervised.

Supervised learning requires input and desired output value for the neural network. This desired output is then used to compare with the output that is computed by the neural network, giving the network a score. A training regimen is formed using the score to improve the neural network.

Unsupervised on the other hand does not require desired output, only feeding inputs into the neural network. Without any outside help, the network has to make sense of the inputs. However, this is not as successful as supervised learning and for this project only supervised learning will be applied (Maind, Wankar 2014).

### **Training Error**

The error for a neural network represents the difference between the desired output and output calculated by the neural network. There are several formulas each having a task of figuring out this error. As the neural network is trained with the intentions of minimizing this error.

### **Model Overfitting**

Model Overfitting is a problem encountered when the trained neural network is trained too well having a 100% accuracy. It sounds like it would be the best neural network, yet it is a massive problem. The neural network will function perfectly with data provided to train the neural network. However, when the neural network encounters new information, it will not fit at all giving unexpected results. The most common way of solving this issue is the use of large training data and testing the neural network with extra data that has not been used to train it (McCaffrey 2013).

There are various ways to train a neural network, but for this project, three techniques that come to mind are BP, PSO, and GA. These techniques will all be used to change the weights and bias of the neural network to produce an output similar to the desired output. All techniques have maximum epochs, where it repeats the process, each time getting closer to having the minimum error possible. Additionally, each technique has "free parameters" that have to be supplied. These free parameters can make or break the neural network as changing them can have drastic effects on the training. Finding values for these free parameters that work beautifully to train the neural network can require some try and error (McCaffrey 2013).

### **Deep Learning**

The hot topic of deep learning is a relatively straightforward concept. Instead of having one hidden layer, there will be multiple hidden layers, each layer categorising the information, refining it and passing it along to the next. Deep learning lets a machine use this process to represent a hierarchical representation (Schmidhuber 2014). For example, in a facial recognition system, the first layer might look for simple edges. A layer up might look for a

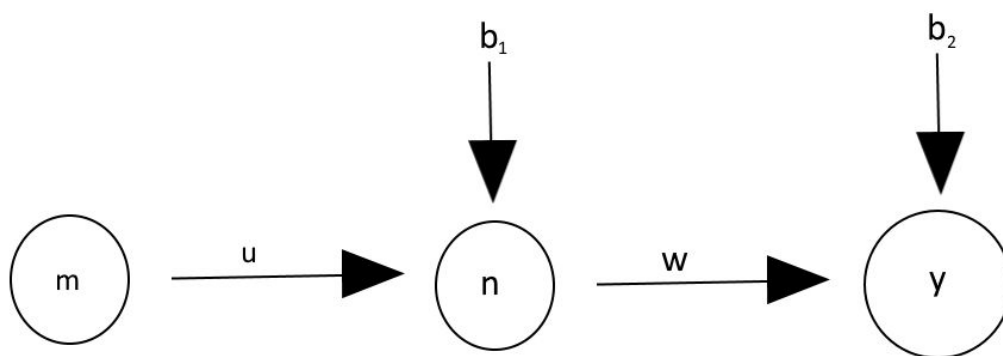
collection of edges, followed by a layer identifying features such as eyes, nose, etc. (Cireşan, Schmidhuber 2013) Since the project is a simple training to fit numbers and outputs are expected as numbers. Deep learning will not be used since there is no need for it in the project.

### Back propagation

Back propagation technique is the standard for training a neural network and is present in most research papers alongside neural networks. To explain the inner workings of BP a training error needs to be computed by going through the neural network once. Then neural networks can be propagated backwards from end to start. Using this process, the derivative (rate of change) of the error for each neuron and synapse can be computed (Werbos 1990).

The derivative error provides information on which direction will minimize the error. Allowing the synapsis weights to be changed slightly in the direction which will minimize the error (Rumelhart, Hinton, Williams 2013).

The reason why small changes are made to the weights instead of just subtracting the error is because it might fit this particular scenario but with a different training data the opposite effects might occur resulting In half of the training data fitting too well and other half not fitting at all.



Using previous research, we can go through an example of back propagation. In the above diagram 'y' being the output node and 'm' being the input node. We can see how back propagation could be used to figure out weight values of 'w' and 'u'.

The derivative error of 'w' needs to be computed so 'w' can be changed slightly towards the direction of minimizing the error. To compute the derivative error of 'w' we can use the following equation.

$$\frac{dE}{dw} = \frac{dE}{dy} A^{-1}(y) \cdot n$$

This equation states that the derivative error of 'w' is equal to the derivative error of 'y' multiplied by derivative activation function of 'y' multiplied by n (Rumelhart, Hinton, Williams 1986).

Since 'y' is the output node we can compute the derivative error of 'y' ( $\frac{dE}{dy}$ ) by taking away the output by the desired output.  $A^{-1}$  is the derivative of activation function that was used in the neural network. Assuming logistic sigmoid was used we can then compute  $A^{-1}(y)$  using the following  $(1-y) * y$ . Lastly the n just represents the value of node 'n'.

After computing the derivative error of 'w' we can change the 'w' weight value using the following.

$$w = w - S \cdot \frac{dE}{dw}$$

In the above equation 'S' is a free parameter supplied by the user. It represents "learning rate". The weight of 'w' synapse is represented by 'w' and the derivative error of 'w' is

represented by  $\frac{dE}{dw}$ .

To update the synapse 'u' it is a bit more complicated. We first have to compute the derivative error of 'n' before computing the derivative error of 'u'. To accomplish this task we can use the following equation to compute the derivative error of 'n'.

$$\frac{dE}{dn} = \sum_{i=0}^L A^{-1}(y_i) \cdot w_i \cdot \frac{dE}{dy}$$

The above equation states that the derivative of the activation function with the value of node 'y' multiplied by the weight of the synapse 'w' multiplied by the derivative error of 'y'. Since y is the output node the derivative error is equal to output - desired output. The sigma sign represents this process needs to be repeated for each weight going out of the node 'n'. Then all of the computed values using this process must be added on to get the final output (derivative error of node 'n'). To explain this process further here are a few steps that can be followed to compute the derivative error of the node 'n'.

Set E to zero

For each weight coming out from current neuron

Set DA to derivative of Activation Function (the node this weight is connected to)

Set W to the value of this weight

Set YE to the derivative error of the node this weight is attached to

E is equal to E + (IA \* W \* YE)

End for

(Lavrenko 2017)

We can compute the derivative error of 'u' using the first equation we used for 'w'. Followed by updating the weights. Since the node 'm' is the input node we can simply ignore it.

The only thing left to compute now is the bias b1 and b2. Looking the previous literature, we can treat these as a dummy node with a weight. Therefore, we can use the same process to compute bias as we used to compute 'w' or 'u'

In addition to computing these weights using derivative errors some techniques can be applied to help get better results such as momentum strategy or weight decay (Wani 2014).

Momentum strategy is a technique that checks the previous derivative error of the weight in the last iteration then adds a fraction of it on to the current weight. Meanwhile weight decay is a technique that removes a fraction of the current weight.

As an example, we can take the synapse 'w' from the above diagram and apply the momentum strategy followed by weight decay assuming the weight for the synapse 'w' has already been computed for this epoch.

To apply the momentum strategy, we can use the following formula

$$w = w + (M * \frac{dE}{dw})$$

An important note from the above formula is that  $\frac{dE}{dw}$  is not the derivative error of 'w' but the last epoch's derivative error of 'w'. Also, the free parameter supplied by the user is represented by 'M'.

To apply the weight decay, we can use the following formula

$$w = w - (D * w)$$

In this weight decay formula 'D' represents the free parameter that has to be supplied by the user. This is a simple formula where the current weight is equal to current weight take away a fraction of current weight.

### Particle Swarm Optimisation

Increasing in popularity in regards to training neural networks the technique known as PSO is very different from BP. PSO is a metaheuristic and uses particles to solve problems. To understand PSO we have to understand these particles and what they are used for.

Each particle in PSO has a memory, in which it holds its current position, current velocity, current error, best position ever found by this particle and best error ever found by this particle (using its position).

Moreover, there are also global variables accessible by every particle. These global variables are as follows best position found by any particle, best error found by any error (Kennedy, Eberhart 2002).



In PSO an initial set of particles are created in a world space for example 2-Dimensional world space requiring position, velocity, best position of each particle and the best position global variables to be setup accordingly. Then a series of steps need to be followed to compute the best position and minimal error.

loop max epochs times

    for each particle

        change velocity depending on current velocity, best position and best swarm position

        update particle position using velocity

        set best error & best position if current particle error is lower

        set best swarm error & best swarm position if current particle error is lower

    end for

end loop

report best swarm position

Every epoch PSO goes through each particle updating its velocity using a formula. Then the updated velocity is used to calculate a new position for the particle by just adding the velocity to the current position (Millonas 1994). Moreover, doing checks if the particle's current position is better than the particles best position if so, it updates the best position to be the current position. Finally, if the particles current position is better than the best swarm position than replace swarm best position by the current particle's position (Rana, Jasola, Kumar 2011).

The following formula is used to compute the velocity of a particle.

$$V_{current} = (I \cdot V_{current}) + (C \cdot R \cdot V_{best}) + (S \cdot R \cdot V_{swarm\ best})$$

I = Inertia Weight (Free Parameter)

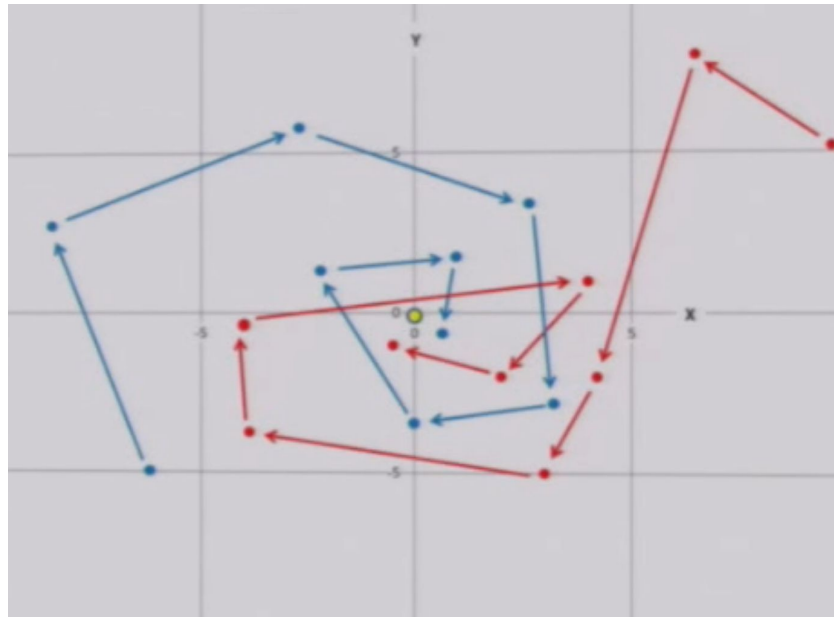
C = Cognitive Weight (Free Parameter)

S = Social Weight (Free Parameter)

R = Random Number

One of the advantages of particle swarm optimisation is, its highly resistant to the free parameters. While with BP changing the learning rate by miniscule amount can cause either beautiful results or your entire BP to fail (McCaffrey 2015).

The below diagram shows a visualisation of 2 particles moving in 2-dimensional world space. Where the yellow dot represents the optimal solution.



Regarding neural networks, PSO can be used to train the neural network in the following manner. The world space where particles will move around in will have X number of dimensions, where X is the total sum of all the weights and biases present in the neural network. For example, an MFNN with 3 input nodes, 1 hidden node and 2 output nodes will contain 5 weights and 3 biases. Therefore, a world space with 8 dimensions will be created where the positions of the particle represent weight and bias values. Moreover, the error of each particle will be the training error of the neural network. To figure out which particle has better position values, error will be used since it is just a number. The lower the error the better the position values are (McCaffrey 1996).

A separate research conducted on using particle swarm optimisation to dynamically create the optimal number of hidden layers. (Abbas, Ahmad, Bangyal 2013) However, dynamic hidden layers using particle swarm optimisation is out of the reach with this project. Going with the simpler approach mentioned above to train the neural network.

## Genetic Algorithm

Genetic algorithms are an optimisation technique used to solve non-linear problems. GA uses concepts from evolutionary biology to search for a global minimum to an optimisation problem. The name Genetic Algorithm comes from the fact they are mimicking evolutionary biology techniques (Turing 1950).

GA works by starting with an initial generation of candidate solutions known as agents. This initial generation will be tested, and subsequent generations evolve from the previous through selection, crossover and mutation (Davis 1991).

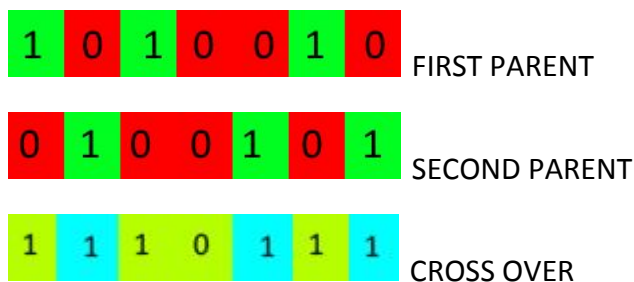
Selection is the process where only the finest agents are selected, discarding all others. Through selection some agents go to the next generation just because they performed well in the first generation. Since these agents performed well, they might be used for crossover.

In the crossover, we use two different agents using both of their data in a mixture to create a new agent. These children agents will be added on to the next generation in addition to the first agents that passed the selection process (Mitchell, Forrest 1993).

The final step is mutation where an agent is taken and has data changed randomly. This avoids falling in the local minimum helping GA explore the solution that might have been missed (Gupta 2012).

Similar to PSO, in neural network GA uses agents, each containing its error and an array of numbers. This array of figures can be represented by the weights and bias. In the world of GA, each agent represents a potential solution with its array of numbers. Furthermore, the agent's error can be computed by using the array of figures for the neural network's weights and bias. This error is also known as the training error, training error for each agent is used to determine whether the agent is selected for the next generation. Generations in neural networks can be transposed to epochs, the number of time this training task needs to be repeated (Giesl, Esponda, Forrest 2001). Then cross over can happen between two agents that are selected for next generation. Regarding neural networks, the crossover can work by using some of weights/bias from the first agent and the rest from the second agent. Applying mutation to an agent is simple, the array containing numbers will have some of its numbers changed slightly with a mutation factor. (Mahajan, Kaur 2013).

Here is an example of cross over. The 2 agents are represented by a binary array. Which can be replaced with a number array containing weights & bias.



The crossover will work by randomly assigning each value to either the first or second parent value in that specific position. Resulting in a randomised mix of the first and second parent.

We can see the cross over node is using part of the first parent and the part from the second, creating a new child. Then a mutation can be applied randomly to change a few numbers. The mutation is done by multiplying a random number with a free parameter, mutation factor. These mutation value are computed for each number in the list of data. For the above example since there are 7 items, seven mutation values will be computed and added on to each value respectively (Forrest, Javornik, Smith, Perelson 1993).

Finishing the training process, we can just use the array of numbers from the finest agent found and plug those numbers into the weights/bias of the neural network.

### 3. Methods

In this section, it will be explained how the project will undergo to reach its conclusion. The specified methodology 'develop and test' will be further explored.

#### 3.1. Methodologies & the Experiment

Develop and test methodology is used because finding a neural network library which is including all three of the techniques with the specifications required for MFNN is no easy task. Therefore, this project will need to undergo a development stage. Using an already constructed example of MFNN, BP, or PSO, each technique will be implemented in the neural network library developed in this project. Followed by implementing GA. GA will not be applied using examples since it is rare to see it, training a neural network and being a simple metaheuristic it will be developed and added to the neural network library.

Although develop and test methodology is used in this project, it is not as complex enough to adhere to this project. In the project, these different techniques (BP, PSO, and GA) will be compared against each other by running several experiments using the developed neural network. Therefore, some form of experimental techniques will be deployed to show the results of the end experiment. The following techniques will be judged as they are trained based on the following three criteria:

- Error on each epoch

- Processing time required for one epoch

- Number of epochs required to reach a suitable error

- Time required to reach 1k epoch

These four values will then be used to describe how efficient is each technique at training the identical neural network. Finally, near the end of the project, the results taken from the previous stage will be inserted into easy to read and detailed graphs. This will allow more analysis of the results to reach a suitable conclusion. Moreover, answering the research question Comparing back-propagation(BP), particle swarm optimisation(PSO), and genetic algorithm (GA) to determine what is the most efficient way to train a multi-layered feedforward neural network.

The neural network will be trained to solve four different datasets from 'UCI machine learning repository.' These datasets are as follows:

- Iris

- Wine

- Breast Cancer Wisconsin (Diagnostic)

- Car Evaluation

These four datasets are chosen for their popularity and known to work with neural networks

giving a good result. These datasets have been used by many times by other researchers and have a large amount of data to work with providing an excellent resource for training the neural network.

### **Resources**

Before starting the project a list of items for the project to succeed are noted. Using a popular search engine all of these resources will be gathered.

This involves using a machine that is able to run all the necessary software required to perform this experiment. However, this is a simple task as most computers today are capable of doing massive computational work.

A program to develop the neural network library using C++ is needed. The program for my choice is VS Code. The reasoning behind choosing VS code is the fact VS code is cross platform. This means writing the code once and it can be run on any machine.

Additionally, some sort of spreadsheet program is required. There are several open source programs that will do the job such as google sheet or open office. For this project google sheet will be used as it is easy to share the document using google sheet, google sheet is an online solution which means that no download is required to run the program.

Dr McCaffrey's work on neural networks includes raw code for a neural network using 1 input layer, 1 hidden layer and 1 output layer. In addition, Dr McCaffrey also provided implementations of BP and PSO.

However, some changes will be applied to the code for it to be more flexible with different data and adding addition features such as the ability to use multiple activation functions. The code will also include the ability to use any of the 4 most commonly used activation functions to provide a reasonable output for each dataset.

The reasoning for using Dr McCaffrey's research is because it is widely used by many people it has been tested and made sure that there are no bugs.

### **3.2. Inner workings of the project**

A neural network library will be constructed using Dr McCaffrey's examples provided in his website. Although the neural network example provided by Dr McCaffrey is in c#. The neural network constructed in this paper will be using C++ since my familiarity with the language, using it as a daily driver. But the same principles will be used to construct the neural network experiment used by Dr McCaffrey.

The neural network will include all four activation functions mentioned above. The reasoning behind this is to encompass all the datasets, so the neural network can work optimally for each dataset. Every activation function has a different output range. Since each dataset has a different type of output value(s) (for example, Boolean, float etc.) different activation function may be used to compute that value.

The neural network library itself is useless unless you can train it. Therefore, different

techniques will be implemented that will train the neural network (BP, PSO, GA). BP is widely used so finding several examples of it being utilised to train a neural network is a simple task. PSO is used less. However, Dr McCaffrey has an all-in depth guide on PSO and implementing it to train the neural network. Again, these examples will be used to build the techniques into the already existing neural network library. Lastly, GA is a special case since there are not many people using it to train the neural networks, it will have to be implemented using research conducted into the field by Stephanie Forrest.

With the developing aspect of the project finished, the next stage is getting results from the developed library. The developed neural network library will be used to create a multi-layered feedforward neural network. This network will be trained using each technique to a given dataset, noting down the error on each epoch as well as the duration of each epoch in milliseconds. Then the program will be let to run until it reaches 1000 epochs. After reaching this epoch the error will be noted. This collected data will then be used to analyse how useful is each technique at training the neural network.

### **Asynchronous Compute**

Normally when a neural network program is developed, it uses multiple cores of the processor to accomplish tasks quickly and in some cases using the graphics card instead of CPU to compute. However, for this project we will stick with a linear coding style where all the operations will occur on a single core. This is due to time limitations and the project being complex as it is already.

### **Free Parameters**

For each training technique, there are free parameters that must be supplied to each technique. To keep the consistency in comparing each technique all of the parameters will be kept the same in all testing. The following free parameters are used for each technique

BP: learning rate (0.05) momentum (0.01) weight decay (0.00001)

PSO: inertia weight (0.729) cognitive weight (1.49445) social weight (1.49445) particles (20)

GA: mutation amount (1.0) agents amount (20)

These free parameters were chosen by looking at other neural network research such as Dr McCaffrey's work and are found to be optimal at giving a minimal error. These free parameters will also be further tested in the testing stage.

### **Data Re-ordering**

Following several other researchers only a section of the data is used for example 80% of the training data is used and on top of that the training data is re ordered to randomize it. Since we are interested more in the training aspect then using the actual neural network 100% of data will be used to train the neural network. In addition to that the re-ordering of the data will be applied before passing the data to each technique to begin the training process. Ensuring better results when training is occurring since the data is in a random order.

## **Data Normalisation**

Before the neural network can be trained using training data. Firstly, we have to normalize the data. This involves setting a range for the data where each input takes a data from 0 to 1. This helps the training process of the neural network. For example, if you have age and income as 2 input nodes. These will have drastically different values. The age might be 25 whereas the income might be 25,000. Since a computer can't distinguish age and income we normalize the data. This can be done by taking the maximum income that a person can have and dividing each income by the maximum income. The same process is done for all other inputs resulting in a value between 0 and 1.

For this project, an already existing normalisation function is used. This function is created by Dr McCaffrey in his attempt to create neural network using BP and can be found on his website. Since Dr McCaffrey's function is working splendidly there is no need to create another.

## **Activation Function**

For this project, only logistic sigmoid activation function is used. The reason for only using logistic sigmoid is Heaviside step does not work with back propagation, SoftMax requires the neural network to be built in a specific way so all of the nodes in a layer are computed simultaneously. Lastly since the data being used in this project only contains positive numbers hyperbolic tangent will hinder the process since its output is between -1 to 1 while logistic sigmoid is from 0 to 1.

## **Training Error**

Training error function will also be taken from Dr McCaffrey's research. Since that particular training error function is widely used by others, the risk of encountering errors will be minimized by the use of pre-existing and tested training error function.

## **3.3. Hypothesis**

Finally, near the end of the project, the results taken from the previous stage will be inserted into easy to read and detailed graphs. This will allow more analysis on the results to reach a suitable conclusion. The hypothesis is that PSO will take the lead converging at the solution rapidly. Lingering being GA will converge to the error slowly. However, if PSO gets stuck in a local minimum error instead of finding a global minimum error, GA will surpass PSO. Meanwhile, BP will be highly unpredictable due to learning rate and momentum (free parameters) having such a significant impact in the learning process.

GA is based on real world natural selection, and if we know anything about natural selection, it is that it takes many centuries to take place. Therefore, we can reach a logical conclusion that GA will take longer than BP and PSO. Since GA adds a mutation, this makes it highly resistant to getting stuck in local minimum error because if it gets stuck theoretically, one agent can be mutated enough to escape and fight for a better minimum which is perhaps the global minimum. Also, Dr McCaffrey points out that back propagation is highly unstable, meaning changing the free parameters by a tiny amount can make drastic changes to how well the neural network is trained. For that reason, BP will be unpredictable working beautifully for one set of datasets and failing to train another dataset.



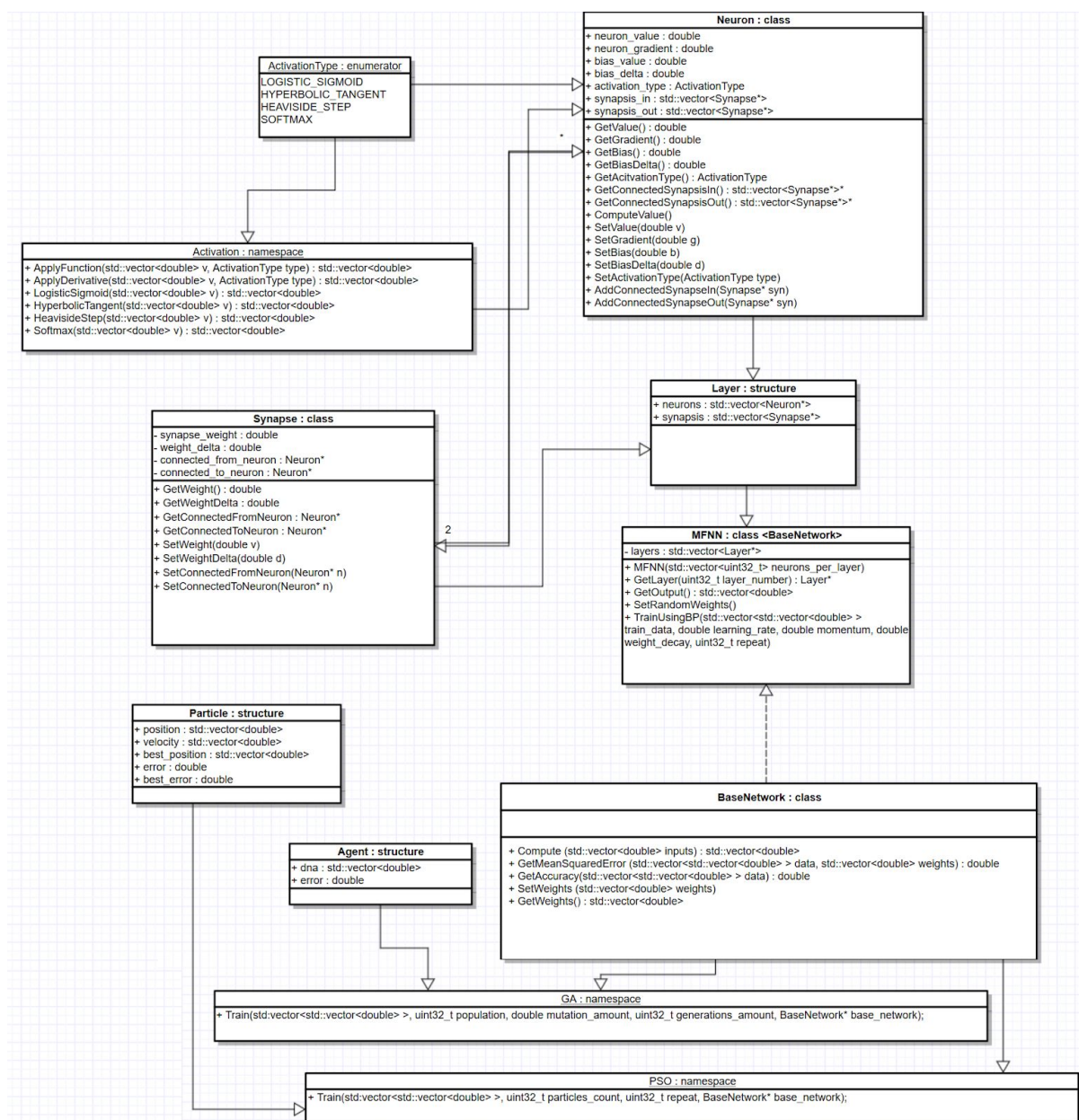
## 4. Implementation & Testing

Since this project is not meant for public use but only used to run once to compute the results of the neural network experiment, a simple console application will suffice.

### 4.1. Class Diagram

Class diagram of the entire project is created before we start programming. This helps reduce the risk of failure since we are not improvising or surprised by something further down the line.

The class diagram consists of a structured overview of the entire solution. This includes all the classes and the variables they will store in their memory. Furthermore, this will also display the methods that will be in each class, showing the purpose of each class.



In the above class diagram, the name of the item is written followed by a colon and then the type of the item. For example “ActivationType : enumeration” represents an enumeration called ActivationType. The item containing a greater and less than sign with another item’s name inside it, indicates that it is inheriting all of its features. In this case, the MFNN class is inheriting from BaseNetwork.

The arrowed lines represent which item requires other items to function properly. Some lines have numbers written on them showing how many of the other items are required. For example, a neuron might have many synapses going into it and many synapses going out from it. Therefore, it has \* next to line going to synapse. While the synapse can only have 2 neurons. The one synapse is going to and the one neuron where the synapse is coming out from. For that reason, it has the number 2 written on the line. Lastly the dotted lines represent inheritance.

In the diagram, the functions and variables are separated via a line. The text after the colon for variables represents the type of variable. For functions, it represents the type of output the function will produce. However, if a function does not have any text, it will simply not return anything. Furthermore, the + and – signs next to each function and variable represent their accessibility by others. The – signed items are private, this means that they cannot be accessed by anything outside the class or structure they’re in. However, the + sign indicates that the item is public and can be accessed out of the class or structure.

The difference between a class and a structure is that structures default to public variables while class defaults to private. Therefore large data is usually stored in structures. Moreover namespaces are entirely different ball game. They are public functions that only have 1 copy. Meanwhile class and structures are skeletons for each copy of object that will be created using them. Meaning they can have many objects.

This particular layout/structure for the solution was chosen because it is very flexible and adaptive. This means in future if other different types of neural network structures need to be added then, it can be done simply by creating another class rather than re-engineering the already created program.

However, one thing to improve on this structure can be the actual computing of the neuron’s value is done in the network structure class (MFNN). This is so SoftMax can function properly. SoftMax requires all the neurons in a layer to be computed simultaneously. But since SoftMax is not being used in this project there is no need to change already working solution.

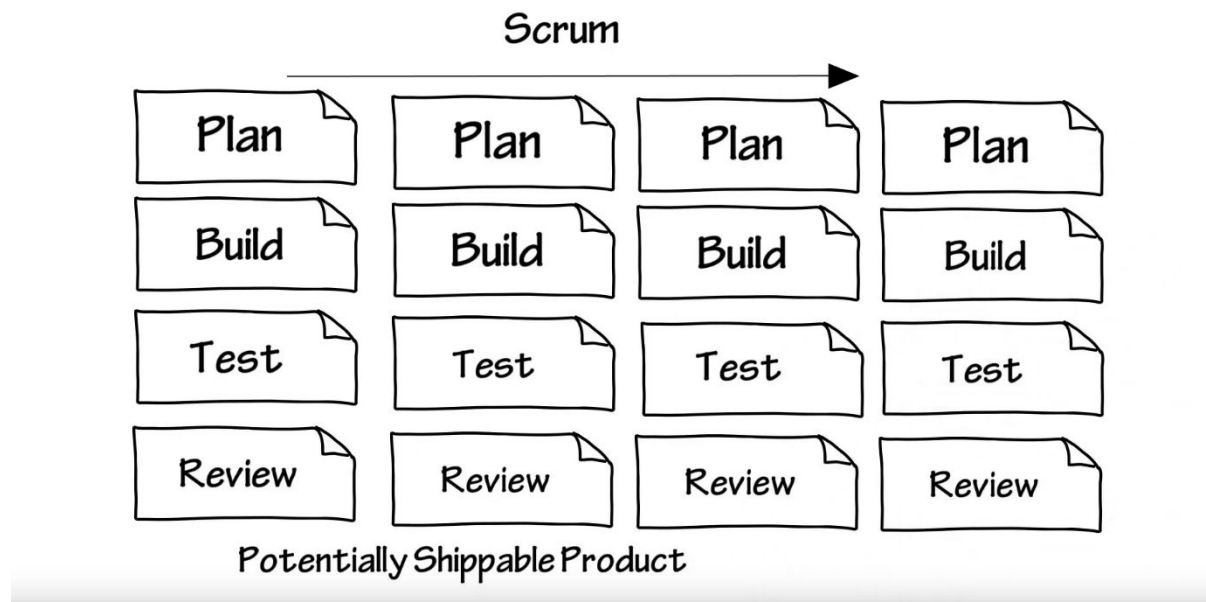
## **4.2. Project Life Cycle**

The project itself starts out with a plan on how the development of the project is done. Using a “scrum” methodology.

The reasoning behind choosing scrum is because it is widely popular and used by big businesses. In addition to that scrum is quite flexible, if a problem arises then it can be dealt with quickly not hindering the entire development stage.

Scrum works in sprints; each sprint involves creating a plan to implement a small feature set.

In this project that can translate to planning how activation functions are implemented. Then this feature set is implemented into the project. Followed by testing the developed features and ending the sprint with a final review of the implemented feature set. This process is repeated until the project is completed.



In this project, the sprint planning takes place just before the sprint starts. At this stage, the class diagram is looked over. A plan is then formed for how the code will be developed and implemented into the solution. This involves having a good understanding of computer programming.

A sprint backlog is created outlining the requirements and feature(s) to complete in this sprint. Then the sprint takes place which can take 1-3 days for this project. During that time, the code is developed and implemented into a solution. In addition, a note is taken of the current progress of the project each day.

Finally, the sprint is finished with testing and reviewing the features implemented into the solution.

This processes leaves us with a potentially completed solution. Moreover, the same process occurs until all the necessary features are developed. After implementing the entirety of the features that are needed for this project to succeed then several tests are carried out to ensure the product is working as intended. If there are issues with the solution after it is feature complete then another sprint is added on with the intention of fixing that specific problem.

### 4.3. Implementation

The implementation of the project is done in a structured manner, where each section is implemented and then test. This allows the bugs to be isolated and fixed immediately as the arise. In addition to that a set of tests were carried out separately after completing the entire solution.

A new project in VS code is setup and tested by just outputting “Hello World” to the console. This allows a quick overview if anything is not functioning as intended.

### Activation Functions

The first step is to implement the activation functions. A separate class is created for the activation functions and a method for each activation function will be created then tested to ensure its functionality.

```
std::vector<double> Activation::LogisticSigmoid(std::vector<double> v){
    //An array of numbers used to store results in.
    std::vector<double> result;
    //Resize the array to match the parameter supplied by the user.
    result.resize(v.size());
    //go through each number provided by the parameter
    for (uint32_t i = 0; i < v.size(); i++)
    {
        //Apply the logistic sigmoid function to each number
        //and store it in result array.
        result[i] = 1.0 / (1.0 + exp(-v[i]));
    }
    //return the result array.
    return result;
}
```

The above code is implemented for logistic sigmoid. It takes a list of doubles (number with a decimal point) in as a parameter then applies logistic sigmoid to all of the numbers. Finally outputting the result at the end. This function is then tested using several values to see if it outputs the correct value. The other activation functions then use this function as a base.

```
std::vector<double> Activation::HyperbolicTangent( std::vector<double> v ){
    std::vector<double> result;
    result.resize(v.size());
    for (uint32_t i = 0; i < v.size(); i++)
    {
        result[i] = tanh(v[i]);
    }
    return result;
}
```

```
std::vector<double> Activation::HeavisideStep(std::vector<double> v){
    std::vector<double> result;
    result.resize(v.size());
    for (uint32_t i = 0; i < v.size(); i++)
    {
        if (v[i] < 0)
            result[i] = 0;
        else
            result[i] = 1;
    }
    return result;
}
```

```

std::vector<double> Activation::Softmax(std::vector<double> v){
    //Get max value from the list of numbers
    double max = v[0];
    for (uint32_t i = 0; i < v.size(); i++)
    {
        if (max < v[i])
        {
            max = v[i];
        }
    }

    //Determine scaling factor -- sum of exp (each val - max)
    double scale = 0.0;
    for (uint32_t i = 0; i < v.size(); i++)
    {
        scale += exp(v[i] - max);
    }

    std::vector<double> result;
    result.resize(v.size());
    for (unsigned int i = 0; i < v.size(); i++)
    {
        result[i] = exp(v[i] - max) / scale;
    }

    return result;
}

```

Hyperbolic tangent, Heaviside step are very similar to logistic sigmoid. For hyperbolic tangent and Heaviside step only the formula is changing but SoftMax required some addition effort.

In SoftMax the first step is to compute the maximum value in the list of doubles. Then going through all the list of doubles' and applying exponential function ( $e^x$ ) divided by the maximum value. All of these values are added and stored into a variable known as scale.

Then once again we go through each of the doubles in the list and apply exponential function to the value minus the max value. Then dividing the result by scale variable resulting in the final output.

Of course, after finishing the implementation stage of these activation functions all of them must be tested. The testing for the activation is further explained in the testing section of this report.

After realising that everything is computing as intended the derivative functions were also implemented to ensure back propagation works properly. Then these functions were tested using the same testing methodology as the activation functions counterpart.

```

std::vector<double> Activation::LogisticSigmoidDerivative(std::vector<double>
v ){

```

```

        std::vector<double> result;
        result.resize(v.size());
        for (uint32_t i = 0; i < result.size(); i++)
        {
            result[i] = (1 - v[i]) * v[i];
        }
        return result;
    }

    std::vector<double>
    Activation::HyperbolicTangentDerivative(std::vector<double> v){
        std::vector<double> result;
        result.resize(v.size());
        for (uint32_t i = 0; i < result.size(); i++)
        {
            result[i] = (1 - v[i]) * (1 + v[i]);
        }
        return result;
    }
}

```

A note to take here is that SoftMax and Heaviside step are not present. The Heaviside step's derivative cannot be computed therefore this technique cannot be used by back propagation. However, for SoftMax. According to Dr McCaffrey's research SoftMax's derivative is the same as Logistic sigmoid derivative.

## Neuron & Synapse

The neuron and synapse classes are now setup. These classes will not contain many functions but just store values.

```

//stores the current neuron value
double                neuron_value            = 0;
//stores the derivative error of the current neuron (used by BP).
double                neuron_gradient         = 0;
//stores the bias value that will be applied to this neuron
double                bias_value              = 0;
//stores the derivative error of bias for this neuron (used by BP)
double                bias_delta              = 0;
//Stores the activation function that will be applied in this neuron
Activation::ActivationType activation_type     =
Activation::ActivationType::LOGISTIC_SIGMOID;
//Stores a list of pointers. These pointers point to all the synapsis going
//into this neuron
std::vector<Synapse*> synapsis_in;
//Stores a list of pointers. These pointers point to all the synapsis coming
//out of neuron
std::vector<Synapse*> synapsis_out;

```

The neuron stores the following items in its memory:

- Current neuron value.

- Neuron gradient (used for back propagation).
- Current bias value.
- Bias delta (used for back propagation).
- The activation type used by this neuron.
- The list of synapsis going in to this neuron.
- The list of synapsis going out of this neuron.

These are all setup up as private variables and then getters & setters are setup for each variable.

The next step is to develop the synapse class, which is similar in some ways to the neuron class as it is only used to store resources but does not compute anything.

```
//Stores the current weight of this synapse
double          synapse_weight      = 0;
//Stores the derivative error of this synapse
double          weight_delta        = 0;
//Stores a pointer a neuron to which this neuron is coming out from.
Neuron*         connected_from_neuron = NULL;
//Stores a pointer a neuron to which this synapse is going into
Neuron*         connected_to_neuron  = NULL;
```

The synapse stores the following items in its memory:

- The current synapse weight.
- Synapse weight delta (used for back propagation).
- The neuron this synapse is going out from.
- The neuron this synapse is going to.

Again, all the variables are set up as private and then getters & setters are setup for each variable.

Rather than the neural network computing each neuron's value. It was thought to be easier to implement if each neuron computes its own value. Then the neural network just calls each neuron telling it to compute its value in the correct order. This allows for better testing and if there is a problem with the code, it can be easily resolved.

Therefor a "ComputeValue" function was added to the Neuron class

```
void Neuron::ComputeValue()
{
    //Sets the current value to zero.
    neuron_value = 0;
    //Goes through each synapse connected to this neuron that
    //is feeding into this neuron.
    for (uint32_t i = 0; i < synapsis_in.size(); i++)
    {
        //The weight of the synapse and the value of the node that synapse
        //is coming out from are multiplied then added to the value of this
        //neuron
    }
}
```

```

        neuron_value += synapsis_in[i]->GetWeight() *
synapsis_in[i]->GetConnectedFromNeuron()->GetValue();
    }
    //The bias value is added to the neuron value.
    neuron_value += bias_value;
    //An activation method is applied to this neuron
    neuron_value = Activation::ApplyFunction({ neuron_value },
activation_type)[0];
}

```

As described previously the neuron gets its value by getting the weights of all the connected synapse to that neuron that's value is being computed. Then those weights are multiplied by the previous nodes the synapsis are coming from. Followed by adding on a bias then applying the activation functions.

Notice the neuron value is set to 0 before computing its value. That is there to ensure that previously computed results do not interfere with current computation.

### Multi-layered Feed-Forward Neural Network

After finishing the neuron and synapsis class, a MFNN class needs to be constructed that creates a multi-layered feed-forward neural network. In the MFNN class a separate structure was created that can hold an array of neurons and array of synapsis. This will represent each Layer in MFNN.

```

struct Layer
{
    //list of neurons in a layer
    std::vector<Neuron*>    neurons;
    //list of synapsis in a layer
    std::vector<Synapse*>    synapsis;
};
//list of layers
std::vector<Layer*>        layers;

```

In the MFNN class there is a list of layers, where each layer contains a list of neurons class object and synapsis class object. The first layer will represent the first layer's neurons and the synapsis connecting first and second layer.

For the output layer, the synapsis will be simply be zero since no other layer is connected from the output layer.

When a MFNN object is created, it takes a list of integers as a parameter. These parameters represent the layers and the number of nodes per layer this MFNN will have.

For example: {3, 7, 4}

Will create a MFNN with 3 input nodes, 7 hidden nodes and 4 output nodes. In total having 3 layers.



This kind of structure allows for vast amounts of flexibility in creating the MFNN. Where the user can define the number of layers and nodes that need to be present and the neural network creates that.

When a MFNN class object is created, it has to setup the list of layer objects. This is done in the following manner.

```
layers.resize(neurons_per_layer.size());
//Goes through each layer and the neurons/synapsis inside the layer then
initializes them by calling the constructor.
for (uint32_t i = 0; i < neurons_per_layer.size(); i++)
{
    //Make sure there is no layer with 0 nodes
    if (neurons_per_layer[i] == 0)
    {
        std::stringstream layer_id; layer_id << i;
        throw std::runtime_error("ERROR [MFNN]: layer " + layer_id.str() + "
has 0 nodes inside it. A layer cannot have 0 nodes.");
    }

    //Setup Neurons
    layers[i] = new Layer();
    layers[i]->neurons.resize(neurons_per_layer[i]);
    for (uint32_t j = 0; j < neurons_per_layer[i]; j++)
    {
        layers[i]->neurons[j] = new Neuron();
    }
    //Setup Synapsis
    if (i + 1 < neurons_per_layer.size())
    {
        layers[i]->synapsis.resize(neurons_per_layer[i] * neurons_per_layer[i
+ 1]);
        for (uint32_t j = 0; j < neurons_per_layer[i + 1]; j++)
        {
            for (uint32_t k = 0; k < neurons_per_layer[i]; k++)
            {
                uint32_t convert_id = (j * neurons_per_layer[i]) + k;
                layers[i]->synapsis[convert_id] = new Synapse();
            }
        }
    }
}
```

First, we change the size of layers then for each layer changing the size of neurons and synapsis accordingly. Then each neuron and synapse is initialised with default weights, bias and values (0).

After setting up the layers we need to create pointers to and from synapsis & neurons. This will allow for easier computational. The job of these pointers is to connect the synapsis and neurons. Since right now we have no way of knowing which synapse connects to which

neurons.

```
//Connect Neurons & Synapsis
for (uint32_t i = 0; i < neurons_per_layer.size() - 1; i++)
{
    for (uint32_t j = 0; j < neurons_per_layer[i + 1]; j++)
    {
        for (uint32_t k = 0; k < neurons_per_layer[i]; k++)
        {
            //This is a formula used to order the synapsis so a specific
            synapse can be
            //extracted later on in the process.
            uint32_t convert_id = (j * neurons_per_layer[i]) + k;
            //Set synapsis pointers and neuron pointers to correct values
            //making sure they can access each others values and are
            connected.

            layers[i]->synapsis[convert_id]->SetConnectedToNeuron(layers[i+1]->neurons[j])
            ;

            layers[i]->synapsis[convert_id]->SetConnectedFromNeuron(layers[i]->neurons[k])
            ;

            layers[i]->neurons[k]->AddConnectedSynapseOut(layers[i]->synapsis[convert_id])
            ;

            layers[i+1]->neurons[j]->AddConnectedSynapseIn(layers[i]->synapsis[convert_id]
            );
        }
    }
}
```

Connecting the neuron with synapsis leads us into the next step creating a function that can compute this MFNN. This has to be done in a certain order, input layer, hidden then output layer. A Compute function is setup into MFNN to do this exact job.

```
std::vector<double> MFNN::Compute(std::vector<double> inputs){
    //Insure inputs provided are valid
    if (inputs.size() != layers[0]->neurons.size())
    {
        throw std::runtime_error("ERROR [Compute]: Input layer does not match
input data provided.");
    }
    //Stup Input by assigning the values provided to the first layer
    //of MFNN.
    for (uint32_t i = 0; i < layers[0]->neurons.size(); i++)
    {
        layers[0]->neurons[i]->SetValue(inputs[i]);
    }
    //Compute Values by going through each neuron and calling the compute
```

```

function
    for (uint32_t i = 1; i < layers.size(); i++)
    {
        for (uint32_t j = 0; j < layers[i]->neurons.size(); j++)
        {
            layers[i]->neurons[j]->ComputeValue();
        }
    }

    //output the last layer in the MFNN
    return GetOutput();
}

```

This function starts off by ensuring the input provided by the user through a parameter matches the input layer. Then the inputs provided are substituted for the input nodes values. Followed by going through each node in each layer in order and computing its value. Then we output the output layer nodes values in a list of numbers.

As mentioned above Dr McCaffrey's mean squared error function was used to compute the error for each technique. The following function works by computing the difference between desired output and output. Then the resulting value is squared and added on to a sum. This indicates the larger the training data the bigger the error will be, since we are adding on the error from each result.

```

double MFNN::GetMeanSquaredError(std::vector<std::vector<double> > data,
std::vector<double> weights){
    //Error Checking
    if (data.size() <= 0)
        throw std::runtime_error("ERROR [GetMeanSquaredError]: could not
locate data.");

    //Get the size of input & output layer to ensure data matches it.
    uint32_t    input_layer_size    = layers[0]->neurons.size();
    uint32_t    output_layer_size   = layers[layers.size() -
1]->neurons.size();
    if (data[0].size() != input_layer_size + output_layer_size)
        throw std::runtime_error("ERROR [GetMeanSquaredError]: training data
does not match neural network");

    //Set the current weights to equal to the weights provided through
//the parameter
    SetWeights(weights);

    //Setup input and desired output variables.
    std::vector<double> xValues(input_layer_size); // Inputs
    std::vector<double> tValues(output_layer_size); //Outputs

    //Go through each training data sent from the "data" parameter
    //Set sum squared error to 0
    double sum_squared_error = 0.0;

```

```

    for (uint32_t i = 0; i < data.size(); ++i)
    {
        //Extract data from "data" parameter and store it in the
        xValues/tValues variables.
        std::copy(data[i].begin(), data[i].begin() + input_layer_size,
        xValues.begin());
        std::copy(data[i].begin() + input_layer_size, data[i].begin() +
        input_layer_size + output_layer_size, tValues.begin());

        //Get the output value computed by the neural network
        std::vector<double> yValues = Compute(xValues);
        //Go through each node and add the (computed value - desired
        value)^2
        //to the sum squared error
        for (uint32_t j = 0; j < yValues.size(); ++j)
            sum_squared_error += ((yValues[j] - tValues[j]) * (yValues[j] -
            tValues[j]));
    }

    //Return the sum squared error
    return sum_squared_error;
}

```

In addition to Mean squared error Dr McCaffrey's accuracy function is also implemented to give more useful data for further analysis.

```

double MFNN::GetAccuracy(std::vector<std::vector<double> > data){
    //create 2 variables that will be used to determine if the value
    //computed is accurate.
    uint32_t    correct            = 0;
    uint32_t    wrong              = 0;
    //Get the size of output & input layers and store them in a variable
    //for ease of access
    uint32_t    input_layer_size   = layers[0]->neurons.size();
    uint32_t    output_layer_size  = layers[layers.size() -
    1]->neurons.size();
    //Setup variables that will store the input, computed output and desired
    output
    std::vector<double> xValues(input_layer_size); // Inputs
    std::vector<double> tValues(output_layer_size); //Outputs
    std::vector<double> yValues(output_layer_size);

    //Loop through all the data provided
    for (uint32_t i = 0; i < data.size(); ++i)
    {
        //Fill the input, desired output and then compute the actual output
        std::copy(data[i].begin(), data[i].begin() + input_layer_size,
        xValues.begin());
        std::copy(data[i].begin() + input_layer_size, data[i].begin() +
        input_layer_size + output_layer_size, tValues.begin());
    }
}

```

```

yValues = Compute(xValues);

//Check the maximum value of each output node. If the data provided
//Also has the maximum value in that place then the neural network
//predicted it correctly, otherwise the prediction was wrong.
uint32_t max_computed = 0;
uint32_t max_target = 0;
for (uint32_t j = 0; j < output_layer_size; j++)
{
    if (yValues[max_computed] < yValues[j])
        max_computed = j;
}
for (uint32_t j = 0; j < output_layer_size; j++)
{
    if (tValues[max_target] < tValues[j])
        max_target = j;
}
//Add to the list if it is correct or wrong.
if (max_computed == max_target)
    correct++;
else
    wrong++;
}

//Return accuracy by deviding the correct amount by total data.
return (double)correct / (double)(correct + wrong);
}

```

## Back Propagation

Since back propagation is so intertwined with neural networks and needs to be setup specifically on how the neural network is structured. Therefor back propagation is just a function inside the MFNN class.

```

void MFNN::TrainUsingBP(
    //Training data
    std::vector<std::vector<double> >    train_data,
    //learning rate free parameter
    double                                learning_rate,
    //momentum free parameter
    double                                momentum,
    //weight decay free parameter
    double                                weight_decay,
    //The max epochs this should run for.
    uint32_t                              repeat
){

```

The BP function takes in the training data, learning rate, momentum, weight decay, and repeat. The first step is to ensure that the training data matches the inputs and outputs.

Therefore, several checks are done to ensure correct parameters are being passed in.

```
//Error Checking
//Make sures the training data does not equal NULL
if (train_data.size() <= 0)
    throw std::runtime_error("ERROR [TrainUsingBP]: Train data is does not
contain any data");
//Make sures the training data matches the neural network that is being
trained
if (train_data[0].size() != layers[0]->neurons.size() + layers[layers.size() -
1]->neurons.size())
    throw std::runtime_error("ERROR [TrainUsingBP]: train data size does not
match the neural network");

//max epochs cannot be smaller then 1
if (repeat < 1)
    throw std::runtime_error("ERROR [TrainUsingBP]: Repeat must be greater
than 0");
```

Now some variables are setup to ensure BP training goes through.

```
//Setup
uint32_t      repeat_counter      = 0;
uint32_t      input_layer_size    = layers[0]->neurons.size();
uint32_t      output_layer_size   = layers[layers.size() - 1]->neurons.size();
```

These variables are as follows:

- Repeat counter is used to keep a track of the current epoch.
- Input layer size contains the number of nodes present in the input layer.
- Output layer size contains the number of nodes present in the output layer.

Followed by starting a while loop that goes through each epoch. At the very next line the time in milliseconds is noted since that is needed to be noted for this project.

```
while (repeat_counter < repeat)
{
    //stores the current time in "begin" variable. This time is highly
accurate.
    auto begin = std::chrono::high_resolution_clock::now();
```

We can then loop through the entire training data and start the training process.

```
for (uint32_t d = 0; d < train_data.size(); d++)
{
    //Get inputs, desired outputs and computed outputs so the training process can
begin.
    std::vector<double> xValues(input_layer_size); // Inputs
    std::vector<double> tValues(output_layer_size); //Outputs
    std::copy(train_data[d].begin(), train_data[d].begin() + input_layer_size,
xValues.begin());
    std::copy(train_data[d].begin() + input_layer_size, train_data[d].begin() +
```

```
input_layer_size + output_layer_size, tValues.begin());
std::vector<double> yValues = Compute(xValues);
```

First step is to store the current training data being worked on in a separate variable for ease of use. xValues, tValues and yValues are created.

xValues are used to store the input values given to the MFNN

yValues are the output values computed by the MFNN.

tValues are used to store the desired values that should be outputted by the MFNN.

In the next step, all of the neuron's derivative error is computed from output layer to input layer. This process is done all at once since then we can get the derivative error of each synapse more easily than having to go through an extra hoop.

```
//Compute gradient for each neuron from output layer to input layer.
for (uint32_t i = layers.size() - 1; i > 0; i--)
{
    for (uint32_t j = 0; j < layers[i]->neurons.size(); j++)
    {
        //Apply the derivative of the activation function
        double derivative = Activation::ApplyDerivative({
layers[i]->neurons[j]->GetValue() },
layers[i]->neurons[j]->GetActivationType())[0];
        double sum = 0.0;
        if (i == layers.size() - 1) //If Output layer
        {
            //If it is the output layer the derivative error is desired -
actual output
            sum = (tValues[j] - yValues[j]);
        }
        else //Else other layer
        {
            //Go through each synapse going out from this neuron
            for (uint32_t k = 0; k < layers[i + 1]->neurons.size(); k++)
            {
                //Multiply the synapse weight by the node gradient to which
this synapse is going to.
                uint32_t convert_id = (k * layers[i]->neurons.size()) + j;
                double x = layers[i + 1]->neurons[k]->GetGradient() *
layers[i]->synapsis[convert_id]->GetWeight();
                //The resulting value is added to a sum
                sum += x;
            }
        }
        //The gradient of the node is then derivative multiplied by the sum.
        layers[i]->neurons[j]->SetGradient(derivative * sum);
    }
}
```

```
}
```

These derivatives are stored in the neuron's class object in the variable known as "neuron\_gradient". Now the final process of updating weights and bias can begin.

```
//Update Weights for each synapse
for (uint32_t i = 0; i < layers.size() - 1; i++)
{
    for (uint32_t j = 0; j < layers[i]->synapsis.size(); j++)
    {
        //get delta by multiply learning rate (free parameter) with synapse
        gradient
        //and then multiplying that by neuron value.
        double delta    = learning_rate *
layers[i]->synapsis[j]->GetConnectedToNeuron()->GetGradient() *
layers[i]->synapsis[j]->GetConnectedFromNeuron()->GetValue();
        //Add the delta to the current synapse weight
        double weight    = layers[i]->synapsis[j]->GetWeight();
        weight            += delta;
        //add momentum multiplied by previous delta to the current weight.
        weight            += momentum * layers[i]->synapsis[j]->GetWeightDelta();
        //multiply weight decay by current weight and then subtract the result
from
        //current weight
        weight            -= (weight_decay * weight);
        //Update all values
        layers[i]->synapsis[j]->SetWeight(weight);
        layers[i]->synapsis[j]->SetWeightDelta(delta);
    }
}

//Update Biases
for (uint32_t i = 1; i < layers.size(); i++)
{
    for (uint32_t j = 0; j < layers[i]->neurons.size(); j++)
    {
        double delta    = learning_rate * layers[i]->neurons[j]->GetGradient()
* 1.0;
        double bias     = layers[i]->neurons[j]->GetBias();
        bias             += delta;
        bias             += momentum * layers[i]->neurons[j]->GetBiasDelta();
        bias             -= (weight_decay * bias);
        layers[i]->neurons[j]->SetBias(bias);
        layers[i]->neurons[j]->SetBiasDelta(delta);
    }
}
```

To update the bias and weights the same process takes place. First the learning rate (free parameter) gets multiplied by the neuron gradient already computed in the previous step.



The result of these 2 numbers multiplied is then stored in a variable called delta. This delta is then added on to the value of the bias/weight.

Since we are using momentum strategy, previous delta that was computed by BP is multiplied by momentum (free parameter) and then the resulting value is added on to the value of bias/weight.

In addition to the momentum strategy weight decay is also tagged on to the BP. Weight decay is applied by multiplying weight decay (free parameter) by the current bias/weight. Then the resulting value is subtracted from the current bias/weight.

Before stepping into the next epoch current time in milliseconds is noted again. Then that time is subtracted by the time noted at the start of the epoch.

```
//get current time and subtract it from the time noted in "begin" variable
auto elapsed_secs = std::chrono::high_resolution_clock::now() - begin;
//Convert time to microseconds
long long microseconds =
std::chrono::duration_cast<std::chrono::microseconds>(elapsed_secs).count();
```

Finally, an output is printed on the screen on the current progress before continuing the same process for the next epoch

```
//Output text to the console.
std::cout << repeat_counter << " " << GetMeanSquaredError(train_data,
GetWeights()) << " " << microseconds << std::endl;
//Go to next epoch
repeat_counter++;
```

### Particle Swarm Optimisation

A separate class for PSO was created. Including all the techniques in a single class is possible but the script would become massive and will issues mending mistakes. In that case it was decided to section of PSO and GA.

```
void PSO::Train(
    //training data
    std::vector<std::vector<double> > train_data,
    //amount of particles to use for training
    uint32_t particles_count,
    //max epochs this training should be repeated for
    uint32_t repeat,
    //A pointer to the neural network that
    //needs to be trained.
    BaseNetwork* base_network
)
```

PSO takes training data, the number of particles, max number of epochs to repeat and the neural network PSO should be training. First stage of any function is error checking to make sure correct parameters were passed in.

This is a crucial step because if the wrong data is passed in. This neural network will be able to tell the user and give an acceptable output letting the user know training data and the neural network does not match at the very beginning of the testing phase.

```
//Error Checking
//There cannot be less than 1 particle
if (particles_count < 1)
    throw std::runtime_error("ERROR [PSO::Train]: Particle count must be
greater than 0");
//max epochs cannot be smaller than 1
if (repeat < 1)
    throw std::runtime_error("ERROR [PSO::Train]: Repeat must be greater than
0");
//The pointer to the neural network cannot be equal to NULL
if (base_network == NULL || base_network == nullptr)
    throw std::runtime_error("ERROR [PSO::Train]: BaseNetwork cannot be
NULL");
//Make sure the training data does not equal NULL
if (train_data.size() <= 0)
    throw std::runtime_error("ERROR [PSO::Train]: Train data is does not
contain any data");
```

In the next step the variables for PSO are setup.

```
//Setup min and max position for particles
double MIN = -10.0;
double MAX = +10.0;
//Setup free parameters
double inertia_weight = 0.729;
double cognitive_weight = 1.49445;
double social_weight = 1.49445;
//Store current weights and current weight length in a variable
std::vector<double> current_weights = base_network->GetWeights();
uint32_t weights_length = current_weights.size();
//setup epoch counter
uint32_t repeat_counter = 0;
//Setup variables to be used as random number generators
double r1 = 0; //Random Number 1
double r2 = 0; //Random Number 2
//Setup global variables (best global position, best global error)
double best_global_error =
std::numeric_limits<double>::max();
std::vector<double> best_global_position(weights_length);
//create a sequence variable for randomizing which particle is processed
first.
std::vector<uint32_t> sequence(particles_count);
//create a swarm of particles.
std::vector<Particle> swarm(particles_count);
```

These variables are as follows:

- Minimum position in all dimensions the particle can have (MIN).
- Maximum position in all dimensions the particle can have (MAX).
- Inertia weight is a free parameter supplied by the user.
- Cognitive weight is a free parameter supplied by the user.
- Social weight is a free parameter supplied by the user.
- Current weights refer to the currently set weights & bias in our neural network.
- Weights length contains an integer value which is equal to the size of current weights.
- Repeat counter is used to monitor each epoch and keeps a track on which epoch the program currently is.
- r1 & r2 are random number generators. These variables will contain a random number upon use.
- Best global error is the minimum global error produced by the swarm.
- Best global position is the best sets of weights and bias produced by the swarm.
- Sequence is used to randomize the order of the particles that are processed to give more optimal training scenario.
- Lastly swarm contains a list of particles that will be used for the PSO training.

In this implementation particles are setup in a separate class. This particle class does not contain any functions but is only used for storing necessary variables for the particles.

```
struct Particle
{
    //Particle position
    std::vector<double>    position;
    //Particle velocity
    std::vector<double>    velocity;
    //Particle best position found
    std::vector<double>    best_position;
    //Particle error
    double                error;
    //particle minimum error found
    double                best_error;
};
```

As mentioned above the particle stores a position, velocity, best position, error and best error that will be used by the PSO class to train the MFNN.

The first step in training the neural network is setting up the particles. Each particle is assigned a random position & velocity. Then the particles error is computed. Since this is the only position particle was ever in. Best position and best error are also set to the currently calculated error and position. This process is repeated for the number of particles.

```
//Setup all particles
for (uint32_t i = 0; i < particles_count; i++)
{
    //Change the size of position and velocity to match weights
    std::vector<double> position(weights_length);
    std::vector<double> velocity(weights_length);
```

```

//Use min and max to create a low and high number for the
//number generator
double low  = 0.1 * MIN;
double high = 0.1 * MAX;
for (uint32_t j = 0; j < weights_length; j++)
{
    //Use random numbers to compute a velocity and position
    velocity[j] = (high - low) * ((double)std::rand() / (double)RAND_MAX)
+ low;
    position[j] = (high - low) * ((double)std::rand() / (double)RAND_MAX)
+ low;
}
//Check the error of the current particle
double error = base_network->GetMeanSquaredError(train_data, position);
//Initialise the particle & set all the variables
swarm[i] = Particle();
swarm[i].position = position;
swarm[i].velocity = velocity;
swarm[i].best_position = position;
swarm[i].error = error;
swarm[i].best_error = error;

//If the particles best error is smaller then best global error
if (swarm[i].best_error < best_global_error)
{
    //Then replace the best global position and error with
    //the particles bbest position and error
    best_global_position = position;
    best_global_error = swarm[i].best_error;
}
}

```

Then the training is started by going through each epoch at a time.

```

//Start training
while (repeat_counter < repeat)
{
    //Used to measure the duration per epoch
    auto begin = std::chrono::high_resolution_clock::now();

```

As soon as an epoch is started the time in milliseconds is noted down. Followed by three new variables that will be used to store position, velocity and error for each particle as they are updated.

```

//Create new variables where the newly computed particle's
//position, velocity and error will be stored
std::vector<double> new_position(weights_length);
std::vector<double> new_velocity(weights_length);
double new_error;

```

Then going through each particle in a for loop we can update the velocity

```
//go through all the particles
for (uint32_t p = 0; p < particles_count; p++)
{
    //Select a particle in the list using random ordering
    uint32_t i = sequence[p];

    //Compute particle velocity
    for (uint32_t j = 0; j < weights_length; j++)
    {
        r1 = (double)std::rand() / (double)RAND_MAX;
        r2 = (double)std::rand() / (double)RAND_MAX;
        new_velocity[j] = ((inertia_weight * swarm[i].velocity[j]) +
                           (cognitive_weight * r1 *
                            (swarm[i].best_position[j] - swarm[i].position[j])) +
                           (social_weight * r2 *
                            (best_global_position[j] - swarm[i].position[j])));
    }
    swarm[i].velocity = new_velocity;
}
```

The velocity as mentioned above is computed using the formula already supplied by other researchers. After computing the velocity, the position is updated but the particle is kept within its max and minimum range.

```
//Compute particle position
for (uint32_t j = 0; j < weights_length; j++)
{
    new_position[j] = swarm[i].position[j] + new_velocity[j];
    //Make sure particle does not go out of bounds.
    //using MIN and MAX variables
    if (new_position[j] < MIN)
        new_position[j] = MIN;
    else if (new_position[j] > MAX)
        new_position[j] = MAX;
}
swarm[i].position = new_position;
```

After computing the position and velocity of the particle we can compute the error by using a premade function in our neural network.

```
//Compute Particle error
new_error = base_network->GetMeanSquaredError(train_data, new_position);
swarm[i].error = new_error;
```

After computing the mean squared error we now have to check if this computed particle error is better than the best particle error and also check if it is better than best global error.

```

//Compare current error with best particle error
if (new_error < swarm[i].best_error)
{
    swarm[i].best_error      = new_error;
    swarm[i].best_position   = new_position;
}

//Compare current error with best global error
if (new_error < best_global_error)
{
    best_global_error        = new_error;
    best_global_position     = new_position;
}

```

Then the time is yet again noted and an output is displayed to the screen, showing the current epoch, best global error and time took for the epoch to complete.

```

//get current time and subtract it from the time noted in "begin" variable
auto elapsed_secs = std::chrono::high_resolution_clock::now() - begin;
//Convert time to microseconds
long long microseconds =
std::chrono::duration_cast<std::chrono::microseconds>(elapsed_secs).count();
//Output text to the console.
std::cout << repeat_counter << " " << best_global_error << " " << microseconds
<< std::endl;

```

## Genetic Algorithm

Just like PSO, GA is created in a separate class. Instead of particles GA has agents therefore a structure of agents is setup.

```

struct Agent
{
    //The weights and bias of a neural network
    //also known as the dna of the agent
    std::vector<double>    dna;
    //The current error of this agent
    double                 error;
};

```

The agent contains a DNA (list of numbers). This DNA consists of weights and bias that are computed by the neural network to give an error. Then that error is stored in the error double.

A training method is setup for training the neural network. This training method takes in several parameters such as:

- Data to train the neural network with.
- The number of agents needed to create for this training.
- The amount of mutation to be applied to each agent.
- A pointer to the neural network that is being trained.

```

void Train(
    //The data to train
    std::vector<std::vector<double> >      train_data,
    //The population of agents that should be initialised
    uint32_t                                population,
    //How much should agents be mutated by
    double                                  mutation_amount,
    //The max epoch or generations training should repeat.
    uint32_t                                generations_amount,
    //Pointer to the network that needs to be trained.
    BaseNetwork*                            base_network
)

```

As with any function the first step is to error check at the very beginning. First, we check to make sure the population is 2 or above. This is due to the fact in GA mating is required and it cannot happen unless there are at least 2 agents. Furthermore, due to mating if there are odd number of agents 1 agent will not be used and for that reason only multiple of 2 population is allowed.

```

//Error Checking
//Ensure there are more then or equal to 2 agents
if (population < 2)
    throw std::runtime_error("ERROR [GA::Train]: Population size too low, must
be greater then or equal to 2 for mating");
//Make sure the population size is a multiple of 2
if (population % 2 != 0)
    throw std::runtime_error("ERROR [GA::Train]: Population size must be a
multiple of 2");
//Make sure mutation is above zero
if (mutation_amount < 0)
    throw std::runtime_error("ERROR [GA::Train]: Mutation amount must be a be
equal or above 0");
//Make sure generations amount is bigger than 0
if (generations_amount < 1)
    throw std::runtime_error("ERROR [GA::Train]: Generations amount must be a
be above 0");
//Make sure the neural network pointer is not NULL
if (base_network == NULL || base_network == nullptr)
    throw std::runtime_error("ERROR [GA::Train]: Base Network cannot equal
NULL");
//Make sure the training data does not equal NULL
if (train_data.size() <= 0)
    throw std::runtime_error("ERROR [GA::Train]: Train data is does not
contain any data");

```

Other parameters such as mutation amount, generations amount, base network and train data are also checked. In GA if the mutation amount cannot be negative. Followed by generation amount that must be bigger than 0. Generation amount is the max epochs this training needs to repeat and if it is zero than no training occurs. Then the last 2 checks are similar to previous training methods checking the base network is not null and there is a

training data passed in.

After completing the error checking, the setup for GA can proceed.

```
//Store current weights & bias of the neural network and
//the length of those weights/bias for easy access
std::vector<double>    current_weights      = base_network->GetWeights();
uint32_t              weights_length      = current_weights.size();
//Setup a epoch counter that keeps track of current epoch
uint32_t              repeat_counter      = 0;
//Stores the best agent with the minimum error
uint32_t              best_agent          = 0;
//2 variables that are used to compute random numbers
//so the range of the random number generator is not too large.
double                low                 = -0.1;
double                high                = +0.1;
//This is used to change the order in which agents are processed.
std::vector<uint32_t>  sequence(population);
//A list of agents that will be used to train the neural network
std::vector<Agent>     agents(population);
//Some extra variables to help with the mating processes
std::vector<double>    w1(weights_length); // Used for mating
std::vector<double>    w2(weights_length); // Used for mating
```

These variables are used for the following job:

- Stores the current weights and bias values of the neural network being trained.
- Stores the size of weights and bias values of the neural network being trained.
- This variable is used to keep track of the current epoch.
- “best\_agent” stores the id of the agent that has the minimal error out of all the agents.
- This variable is the minimum value that can be generated by the random value generator.
- Just like the above variable this variable stores the maximum value that can be generated by a random value generator.
- The same as PSO this sequence is used to re order the agents so they are processed in a specific order.
- A list of agents that will be used by GA to train the network.

After setting up the variables we need to initialise the agents. A for loop is setup to go through to go through the entire population. Then each agent is created with a randomly generated DNA. Finally finishing the for loop by computing the error of this agent using the DNA created in the previous step.

```
//For each agent in the list
for (uint32_t i = 0; i < population; i++)
{
    agents[i] = Agent();
    //Change the DNA size to match weights and bias
}
```



```

agents[i].dna.resize(weights_length);
for (uint32_t j = 0; j < weights_length; j++)
{
    //Set each weight and bias to a random value
    agents[i].dna[j] = (high - low) * ((double)std::rand() /
(double)RAND_MAX) + low;
}
//Compute the error using the DNA just created by the agent.
agents[i].error = base_network->GetMeanSquaredError(train_data,
agents[i].dna);
}

```

Now the main loop is started and the training process can begin. First the while loop is created to loop through each epoch followed by noting the current time.

```

while (repeat_counter < generations_amount){
    //Used to measure the duration per epoch
    auto begin = std::chrono::high_resolution_clock::now();

```

The next step is to re-order the agents so the minimal error agents are in front of the list using sequence. This allows us to remove/kill all the weak agents since they will not be needed.

```

//Re arrange arrays with the best being first
//Each agent checks every other agent from left to right
//and if that agent is weaker then the agent it is checking
//they swap places.
for (uint32_t i = 0; i < population; i++)
{
    for (uint32_t i = 0; i < population; i++)
    {
        for (uint32_t j = i+1; j < population; j++)
        {
            if (agents[sequence[i]].error > agents[sequence[j]].error)
            {
                uint32_t temp = sequence[i];
                sequence[i] = sequence[j];
                sequence[j] = temp;
            }
        }
    }
}

```

Reordering the agents leads us to removing the weaker agents from the pack since they are not needed. These agents will then be replaced by the first 2 agents in the back since those agents have the minimal error from the entire pack.

```

//Kill half of the weak population
for (uint32_t i = (population / 2); i < population; i++)
{
    //Replace the agents with the first 2 agents in the pack

```

```

agents[sequence[i]].dna = agents[sequence[i % (population / 2)]] .dna;
}

```

Now each agent is mated with the agent next to it. This process produces 2 children each having weights from 1<sup>st</sup> and 2<sup>nd</sup> parent. The weights are randomized each time so there is no way to tell which child will possess how much from the 1<sup>st</sup> parent DNA and how much from the 2<sup>nd</sup> parent DNA. These children are then replaced by the parents.

```

//Mate Everyone
for (uint32_t i = 2; i < population / 2; i++)
{
    //Go through each weight and bias value
    for (uint32_t j = 0; j < weights_length; j++)
    {
        //Compute a random number
        if (std::rand() % 2 == 0)
        {
            //If the computed random number is even then
            //make the first child's weight be the same as first parent's
            //weight and second child's weight be the same as second parent's
            //weight
            w1[j] = agents[sequence[(i * 2) + 0]].dna[j];
            w2[j] = agents[sequence[(i * 2) + 1]].dna[j];
        }
        else
        {
            //If the computed random number is odd then
            //make the first child's weight be the same as second parent's
            //weight and second child's weight be the same as first parent's
            //weight
            w1[j] = agents[sequence[(i * 2) + 1]].dna[j];
            w2[j] = agents[sequence[(i * 2) + 0]].dna[j];
        }
    }
    //Replace the parents with the child's created using their DNA
    agents[sequence[(i * 2) + 0]].dna = w1;
    agents[sequence[(i * 2) + 1]].dna = w2;
}

```

The next stage is to mutate each child. This is done by computing a random number between the high and low variables then multiplying that number with mutation amount (free parameter). This final value is then added on to the weight. This process is then repeated for each weight and bias.

```

//Mutate Randomly
for (uint32_t i = 2; i < population; i++)
{
    for (uint32_t j = 0; j < weights_length; j++)
    {

```

```

        //Compute a random number between high and low values
        double m = (high - low) * ((double)std::rand() / (double)RAND_MAX) +
low;
        //Apply mutation
        agents[sequence[i]].dna[j] = agents[sequence[i]].dna[j] + (m *
mutation_amount);
    }
}

```

The final stage is to compute the errors of the agents using their DNA including updating the best agent.

```

//Compute errors
for (uint32_t i = 0; i < population; i++)
{
    agents[i].error = base_network->GetMeanSquaredError(train_data,
agents[i].dna);
    //If current agent has a better error then best agent
    //then replace it with the best agent.
    if (agents[best_agent].error > agents[i].error)
    {
        best_agent = i;
    }
}

```

Like all the other techniques we need to compute the time now and take away it from the time stored in begin variable. Followed by converting it to microseconds and outputting the results to the console.

```

//get current time and subtract it from the time noted in "begin" variable
auto elapsed_secs = std::chrono::high_resolution_clock::now() - begin;
//Convert time to microseconds
long long microseconds =
std::chrono::duration_cast<std::chrono::microseconds>(elapsed_secs).count();
//Output text to the console.
std::cout << repeat_counter << " " << agents[best_agent].error << " " <<
microseconds << std::endl;

```

## Training Data

To be able to understand the full extent of this program a basic knowledge on training data must be acquired. The training data is setup in an array, where each item in the array contains another array which contains the actual numbers that will be used as inputs and outputs. In all the training data inputs are listed first then the desired outputs. Here is an example shown below.

```

{5.1,   3.5,   1.4,   0.2,   1,   0,   0, },
{4.9,   3,    1.4,   0.2,   1,   0,   0, },

```

This example is taken from the first 2 data of iris data set. It consists of 4 inputs (5.1, 3.5, 1.4, 0.2) and 3 outputs (1, 0, 0). These outputs are desired outputs.

Furthermore, if a data set includes a category then it is converted into binary value represented in the following manner. The reason why it is written this way instead of a single digit ranging from 0 to 2 is according to Dr McCaffrey's and others research neural network work better if the data is normalised and this type of normalisation has proven to be quite effective with classification data.

-- Iris Setosa                1, 0, 0

-- Iris Versicolour        0, 1, 0

-- Iris Virginica           0, 0, 1

All other numbers (in the above example the inputs) are normalised as the program starts. This is so we can get a clear view of how the data looks in raw code.

#### **4.4. Testing**

As mentioned above the testing phase occurs as each feature is implemented. For this section of the report we will look at each of those testing methods and the final testing done after completing the solution.

Each testing phase is done in 4 stages.

Firstly, the program is compiled with the function/feature implemented that is being tested. This compilation process resolve any minor syntax errors that might creep up. This is a automated process as you will not be able to run the actual program without these syntax errors.

In the next stage the function being tested is given invalid values this is to ensure if by chance any values passed to the function do not make sense. Instead of the program crashing or just simply outputting the wrong results. The program instead shows an error message and stops the user from proceeding further.

Moreover, in the next stage we go through each line of the already implemented feature/function using the built-in VS code debugging tools. This allows us to see how the computer is going through all the code line by line. Furthermore, this also shows some additional information like what are the current values of each variable.

The end stage before concluding the testing. The feature/function is running as normally with a wide range of input values. Then those inputs and the output given by the program are tested to see if they are accurate insuring the function is working properly. If the program is not giving the correct outputs as expected then the 3<sup>rd</sup> stage is deployed again, going through each line of the program to solve the issue.

#### **Activation Functions**

After completing the implementation of the activation functions some testing is done. Some of the results are shown above in the implementation stage. However, in this section we will discuss more on how activation functions are tested in greater detail.

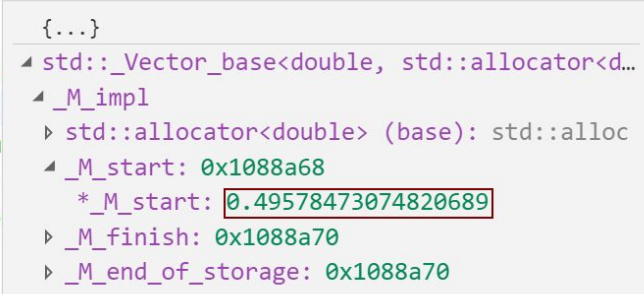
First step of the testing was to compile and ensure the code implemented is working as

intended. Since activation function is just a simple math function there was no complex issue to resolve in this stage of the testing phase. The program seems to run and compile without any issues.

The second stage also does not affect the activation functions since they only accept one type of input. A list of numbers that need to have the activation function applied to them.

This leads us to the second last stage. In this stage, the parameters are checked after they are passed into the function to ensure they have the correct value stored inside them. This is done using the VS code built-in debugging tools. In addition to that the value of the output by the function is also checked as it changes when the code goes through that line.

```
--
41 //Activation Functions
42 std::vector<double> Activation::LogisticSigmoid(
43     std::vector<double> v
44 ){
45     //An array of numbers u
46     std::vector<double> res
47     //Resize the array to m
48     result.resize(v.size())
49     //go through each numbe
50     for (uint32_t i = 0; i
51     {
52         //Apply the logistic sigmoid function to each number
53         //and store it in result array.
54         result[i] = 1.0 / (1.0 + exp(-v[i]));
55     }
56     //return the result array.
57     return result;
58 }
```



The debugger window shows the state of a `std::vector` object. The value `0.49578473074820689` is highlighted with a red box, indicating the current value of the element being processed.

In the above image the value stored in the computer's memory after the activation method has been computed is shown with a red box around it.

This stage of testing involves testing the function by running it thoroughly. For this process, a list of numbers was chosen that would be passed into each function. Then the output from those functions will be compared to the actual value the function gives computed using a calculator on a piece of paper. The output below shows a snippet of numbers that were tested to check if the activation functions are performing correctly.

#### Activation Function Test 1

Applying logistic sigmoid to 1

Result: 0.731059

Applying logistic sigmoid to -1

Result: 0.268941

Applying logistic sigmoid to 0.333333

Result: 0.58257

Applying logistic sigmoid to -0.75

Result: 0.320821

Applying logistic sigmoid to 100

Result: 1

Applying logistic sigmoid to -100

Result: 3.72008e-044

Activation Function Test 2

Applying hyperbolic tangent to 1

Result: 0.761594

Applying hyperbolic tangent to -1

Result: -0.761594

Applying hyperbolic tangent to 0.333333

Result: 0.321513

Applying hyperbolic tangent to -0.75

Result: -0.635149

Applying hyperbolic tangent to 100

Result: 1

Applying hyperbolic tangent to -100

Result: -1

...

---

## Neurons, Synapsis & MFNN

Since Neuron and synapsis are primarily used to just store data there is no need for testing. However, the 1 compute function inside neuron needs to be tested and the MFNN class requires some extensive testing. Therefore, these classes results will be displayed together.

Starting of the testing phase for these classes with again the compiling and checking for any syntax errors. This is important to ensure that the code can be run. In this stage, some minor errors were found where a semi colon was missed or a comma was missing. When

this happens, the compiler outputs the following message.

---

```
NeuralNetwork/Structure/Neuron.h:46:41: error: expected ';' at end of member
declaration
      std::vector<Synapse*>                synopsis_out
```

---

With these compiler issues fixed the program could now start on the next testing phase.

Since neuron and synapsis only store information and do not have functions where you need to pass parameters to there is no need to test those for this part of the testing phase. However, the MFNN does contain a Compute function which takes in a array of numbers. The length of which must match the input size. The first instinct to test this function was to pass in a value that does not align correctly with the input nodes of the MFNN.

---

#### MFNN Compute Test 1

Setup a new MFNN with 2, 1, 1

Sending invalid data of inputs {-5, -5, -5}

ERROR [Compute]: Input layer does not match input data provided.

Sending invalid data of inputs {-5}

ERROR [Compute]: Input layer does not match input data provided.

End of Program

---

This shows that even if the inputs do not align correctly with the input nodes the program does not compute incorrectly but spits out an error. The next test involved in the creation of MFNN. Passing in negative numbers, since there cannot be negative or 0 number of nodes in a layer this should also throw an exception. In addition to that if there is no output layer then another error message should be thrown.

For this test passing a negative number and running the program was impossible due to the use of uint32\_t which can only hold positive numbers. The following output was shown when the program was compiled.

---

```
main.cpp: In function 'int main()':
main.cpp:11:34: error: narrowing conversion of '-5' from 'int' to 'unsigned
int' inside { } [-Wnarrowing]
      MFNN nn = MFNN({-5, 5, 5});
```

---

This part of the test was removed and the program was compiled again.

---

## MFNN Test 2

Creating Neural Network {0, 5, 5}

ERROR [MFNN]: layer 0 has 0 nodes inside it. A layer cannot have 0 nodes.

Creating Neural Network {5}

ERROR [MFNN]: layers must have a size greater than or equal to 2.

## End of Program

---

The program was able to compile this time but displayed error messages indicating that there is something wrong.

“GetAccuracy” and “GetMeanSquaredError” functions are not needed to be tested because the program would not compile if wrong values were passed into those functions. These functions except a list of doubles and a double can be any real number.

In the next testing phase, each function is given correct values and computed step by step. Allowing a clear view of what is happening in the code. The following image shows the variables and objects stored inside the computer’s memory, indicating that everything is working as intended.



```

└─ Locals
  └─ this: 0x74fed4
      BaseNetwork (base): BaseNetwork
      layers
        └─ std::_Vector_base<MFNN::Layer*, std::allocator<MFNN::Layer*> > (bas...
            └─ _M_impl
                └─ std::allocator<MFNN::Layer*> (base): std::allocator<MFNN::Layer*>
                    └─ _M_start: 0x1121390
                        └─ *_M_start: 0x1121430
                            └─ neurons
                                └─ std::_Vector_base<Neuron*, std::allocator<Neuron*> > (base...
                                    └─ _M_impl
                                        └─ std::allocator<Neuron*> (base): std::allocator<Neuron*>
                                            └─ _M_start: 0x1121460
                                                └─ *_M_start: 0x1121258
                                                    neuron_value: 0
                                                    neuron_gradient: 0
                                                    bias_value: -0.091064180425428021
                                                    bias_delta: 0
                                                    activation_type: Activation::LOGISTIC_SIGMOID
                                                    └─ synapsis_in
                                                        └─ synapsis_out
                                                            └─ _M_finish: 0x112146c
                                                                └─ _M_end_of_storage: 0x112146c
                                                                    └─ synapsis
                                                                        └─ _M_finish: 0x112139c
                                                                            └─ _M_end_of_storage: 0x112139c
└─ inputs: {...}
    └─ std::_Vector_base<double, std::allocator<double> > (base): std::_Vect...
        └─ _M_impl
            └─ std::allocator<double> (base): std::allocator<double>
                └─ _M_start: 0x112dad0
                    └─ *_M_start: 1
                        └─ _M_finish: 0x112dae8
                            └─ _M_end_of_storage: 0x112dae8

```

At this stage, Dr McCaffrey's already built neural network was used to quickly test and ensure everything is working properly. To accomplish this task a set of weights were given to this MFNN and Dr McCaffrey's MFNN. Then both were set to compute a few values, upon both giving the same output, it was clear that this projects MFNN was working as intended. Of course, this kind of testing assumes that Dr McCaffrey's neural network is perfect and since it is on public domain viewed by countless people, it is safe to assume that it is working properly. The following output was produced by the testing showing that, given the

correct weights the neural network is functioning as intended by producing the correct outputs.

---

## Neural Network Test

Applying Dr Mccaffrey's already trained weights:

-6.24388, 10, -10, -10, 9.68381, 10, 10, -10, -2.06582, 6.13957, -0.479258, -1.82205,  
-9.98176, 0.199313, -10, -6.62423, 1.68808, 0.123798, -0.676269, -10, 3.41882, 1.20598,  
-1.39019, 5.10291, -2.26213, -1.32441, 6.61477, 0.345021, 0.986209, 1.54915, -2.60002,  
-4.5882, -0.3196, -3.45096, -3.38032, 3.34489, -2.72878, -0.803161, 10, -0.55176, 6.39354,  
10, 9.97832, -6.58292, -5.71315, -0.984417, -6.47856, -0.814072, -8.84718, -1.28892,  
-2.20913, 6.44553, 10, 3.26917, -10, 10, -10, 1.17383, 2.17091, -9.99571, -0.102128,  
-3.03252, -0.856342,

Computing first 10 values from iris data

Input: 5.1, 3.5, 1.4, 0.2

Output: 0.999708, 0.0214834, 3.45328e-012

Desired: 1, 0, 0

Input: 4.9, 3, 1.4, 0.2

Output: 0.999666, 0.0219664, 4.2335e-012

Desired: 1, 0, 0

Input: 4.7, 3.2, 1.3, 0.2

Output: 0.999702, 0.0215868, 3.54735e-012

Desired: 1, 0, 0

Input: 4.6, 3.1, 1.5, 0.2

Output: 0.99969, 0.0221788, 3.67592e-012

Desired: 1, 0, 0

Input: 5, 3.6, 1.4, 0.2

Output: 0.999709, 0.0214753, 3.42423e-012

Desired: 1, 0, 0

Input: 5.4, 3.9, 1.7, 0.4

Output: 0.999708, 0.0215589, 3.43022e-012

Desired: 1, 0, 0

Input: 4.6, 3.4, 1.4, 0.3

Output: 0.999705, 0.0216621, 3.46635e-012

Desired: 1, 0, 0

Input: 5, 3.4, 1.5, 0.2

Output: 0.999704, 0.0216143, 3.49145e-012

Desired: 1, 0, 0

Input: 4.4, 2.9, 1.4, 0.2

Output: 0.999673, 0.022355, 3.97293e-012

Desired: 1, 0, 0

Input: 4.9, 3.1, 1.5, 0.1

Output: 0.999687, 0.0218955, 3.78611e-012

Desired: 1, 0, 0

...

---

### **Back Propagation, Particle Swarm Optimisation & Genetic Algorithm**

With the activation functions, neurons, synapse and MFNN class done the training functions needed to be tested. Testing the 3 training functions can be quite the complex task since neural network in some cases has been dubbed a black box where you don't know what is happening inside but you just give it inputs and get outputs.

Each technique has a train function that needs to be tested. First the compile stage is started making sure the program can be compiled and is able to run. As with any program there are always human mistakes where a comma or a semi colon might be missed. But, there was nothing major to report in this stage.

Next, we have to make sure giving the invalid parameters to these train functions does not lead us with invalid results. For example, if a neural network is setup for wine data set and we pass in iris data set to train it with. Then the training process will have random results. For that reason, checking and making sure invalid parameters produce an error is a must.

BP is tested by passing in the wrong training data with invalid input and output lengths and an error message was produced. However, for PSO & GA it was a different story. Since both PSO & GA are in a separate class that interact with a base class they have no way of knowing the input and output layer size of the neural network. This posed as a problem but it was resolved by adding a check for the data passed into Mean Squared Error Function in the MFNN. Since both techniques need to use that particular function to compute the result. With that done all 3 techniques are capable of handling invalid data but not only that they can output a message if something goes wrong. The following output was taken from this testing phase.

---

Creating Neural Network: {1, 1, 1}

Passing iris data set to BP

ERROR [TrainUsingBP]: train data size does not match the neural network

Passing iris data set to PSO

ERROR [GetMeanSquaredError]: training data does not match neural network

Passing iris data set to GA

ERROR [GetMeanSquaredError]: training data does not match neural network

---

From that output, we can clearly see that no invalid data is allowed to be passed to any training function.

Next step is to go through each function checking the values to make sure they are functioning correctly. In this stage, a massive bug was found with BP that made it not able to function properly. When using BP to train the error always was really big and kept increasing. However, with the help of built in tools in VS code the error was quickly resolved.

The error itself involved computing the derivative error for each node. When computing the derivative error of any other node that is not in the output layer the wrong formula was used to compute the error. The wrong formula is displayed below

```
uint32_t convert_id = (j * layers[i + 1]->neurons.size()) + k;
```

In the project the synapsis are layed out in a ordered fashion and a formula can be used to find each synapse in the layer. However, when using the VS code tools it was found that the synapse was not linking up right which lead to the correction of this formula to.

```
uint32_t convert_id = (k * layers[i]->neurons.size()) + j;
```

This fixed issue and allowed the BP to produce error that was looking appropriate. In this stage, we are only looking for error that is minimizing.

The other techniques were found to be working as intended. With that said we can continue on the next stage of the testing process.

The last stage, testing to ensure each technique is working properly by running the program and compare the results with other already built neural network. First BP was used to train using Dr McCaffrey's already built neural network. After 1 thousand iterations BP seemed to reach above 90% accuracy in iris training set. Using the same configurations, we can then run BP technique to train this project's neural network. The following output was taken from the neural network showing that after 1k epochs it is in fact reaching above 90% accuracy.

---

Creating Neural Network: {4, 7, 3}

Training using iris data for 1k epochs

BP Settings:

Learning Rate: 0.05

Momentum: 0.001

Weight Decay: 0.000001

Neural Network Training Complete, Current Accuracy: 0.966667

---

This is not the end of testing. It proves that we are reaching that accuracy but now the next step is to get the outputs for each epoch showing how the error is minimized. In the following output, we can see that BP is minimizing the error each epoch and working as intended.

---

epoch: 0, accuracy:0.333333

epoch: 9, accuracy:0.666667

epoch: 19, accuracy:0.786667

epoch: 29, accuracy:0.866667

epoch: 39, accuracy:0.893333

epoch: 49, accuracy:0.9

epoch: 59, accuracy:0.92

epoch: 69, accuracy:0.926667

epoch: 79, accuracy:0.933333

epoch: 89, accuracy:0.946667

epoch: 99, accuracy:0.96

---

Similar tests were run on PSO and GA with each outputting results in a similar fashion where the error was getting lower each epoch drastically. The following output proves the point that PSO and GA are working as intended.

PSO Output:

---

Creating Neural Network (4, 7, 3)

Starting training using PSO on Iris Data Set

epoch: 0, accuracy:0.366667

epoch: 9, accuracy:0.826667

epoch: 19, accuracy:0.906667

epoch: 29, accuracy:0.92

---

epoch: 39, accuracy:0.946667  
epoch: 49, accuracy:0.96  
epoch: 59, accuracy:0.966667  
epoch: 69, accuracy:0.986667  
epoch: 79, accuracy:0.98  
epoch: 89, accuracy:0.986667  
epoch: 99, accuracy:0.98

---

#### GA Output:

---

Creating Neural Network (4, 7, 3)  
Starting training using GA on Iris Data Set  
epoch: 0, accuracy:0.6  
epoch: 9, accuracy:0.38  
epoch: 19, accuracy:0.666667  
epoch: 29, accuracy:0.66  
epoch: 39, accuracy:0.706667  
epoch: 49, accuracy:0.946667  
epoch: 59, accuracy:0.806667  
epoch: 69, accuracy:0.886667  
epoch: 79, accuracy:0.9  
epoch: 89, accuracy:0.873333  
epoch: 99, accuracy:0.913333

---

These tests were not just run once, but each technique was tested multiple times. Getting the error to minimize in such a way that each iteration we can reach above 90% accuracy on each technique is not luck but the sign that the techniques are working correctly. Concluding that each technique is working as intended the testing phase was over and now the experiment can commence.

## 5. The Experiment

Before going into the experiment section of this report, some ground rules need to be set, including how the data is stored and passed to the techniques. Furthermore, some clarification is also required on how the techniques are compared and how many times each technique is ran to compute the results.

### 5.1. Preparation

To start of the experiment Iris, breast cancer, wine and car evaluation data sets are downloaded from UCI machine learning repository and stored in each file separately. Then these data sets are organised and optimised for c++ code to read the data quickly and efficiently. Since the neural network only accept double(number) as an input a string cannot be passed. This means a conversion needs to take place from category to double. The conversion is done in the following manner. We replace those category with a binary list of 1s and 0s. Where the length represents the size of a category and for each category only one digit in the list will be set to 1 the rest will all be 0s. According to Dr McCaffrey's experiments this is the best way to pass data in training neural network when using classification data. Here is an example of what the iris classification convert into.

Iris Setosa	1, 0, 0
Iris Versicolour	0, 1, 0
Iris Virginica	0, 0, 1

This leaves us with the actual experiment. The experiment will contain running each technique with each dataset 10 times. Followed by averages taken from those 10 tests and in this section those averages will be compared.

Before the neural network is trained all weights and bias are set to a random value between -0.1 and 0.1. The values that are set are also stored and can be seen in the appendix of this report.

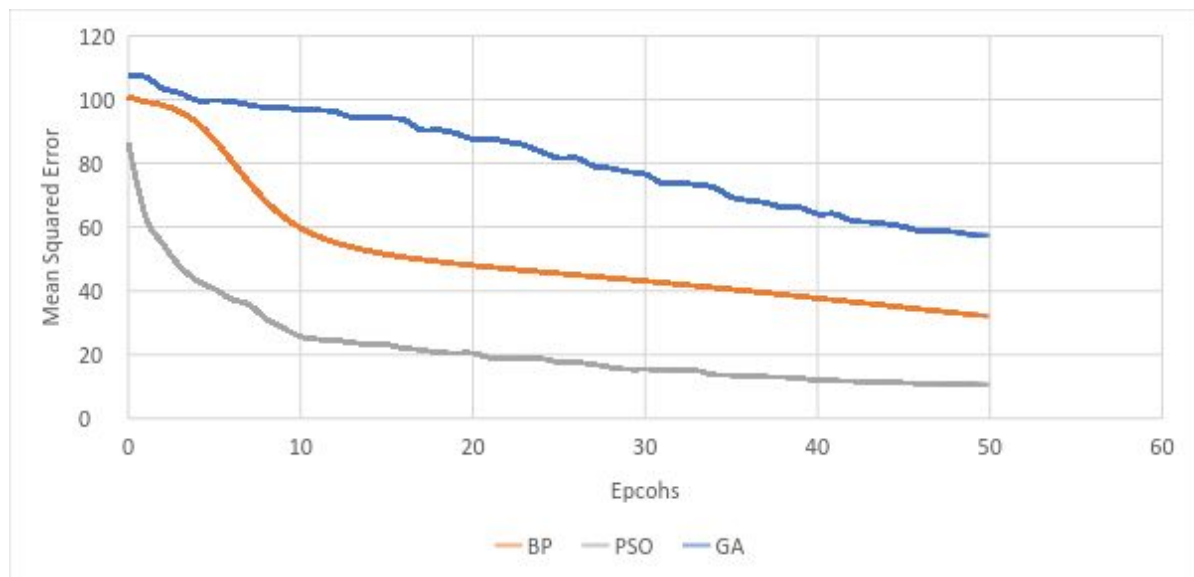
Moreover, for each data set the neural network size does not change when training with different techniques. In the iris data set each technique uses a neural network, which consists of 4 input nodes, 7 hidden nodes and 3 output nodes. For the wine data set a neural network with 13 input nodes, 7 hidden nodes and 3 output nodes is used. Car evaluation data set uses a neural network made up of 21 inputs, 13 hidden and 4 output nodes. Finally, the breast cancer data uses 9 input nodes, 7 hidden nodes and 2 output nodes.

### 5.3. Results, Analysis & Discussion

In this section, we will go through each data set comparing accuracy, error, convergence on the error and repeatability of that score. Getting the same results over and over again is important since it is not guaranteed that each iteration of the test is the same as the one before. Also, there, can be massive changes to each test. Therefore, averages between 10 different tests are taken, and those averages will be used in all the graphs showing the results in this section.

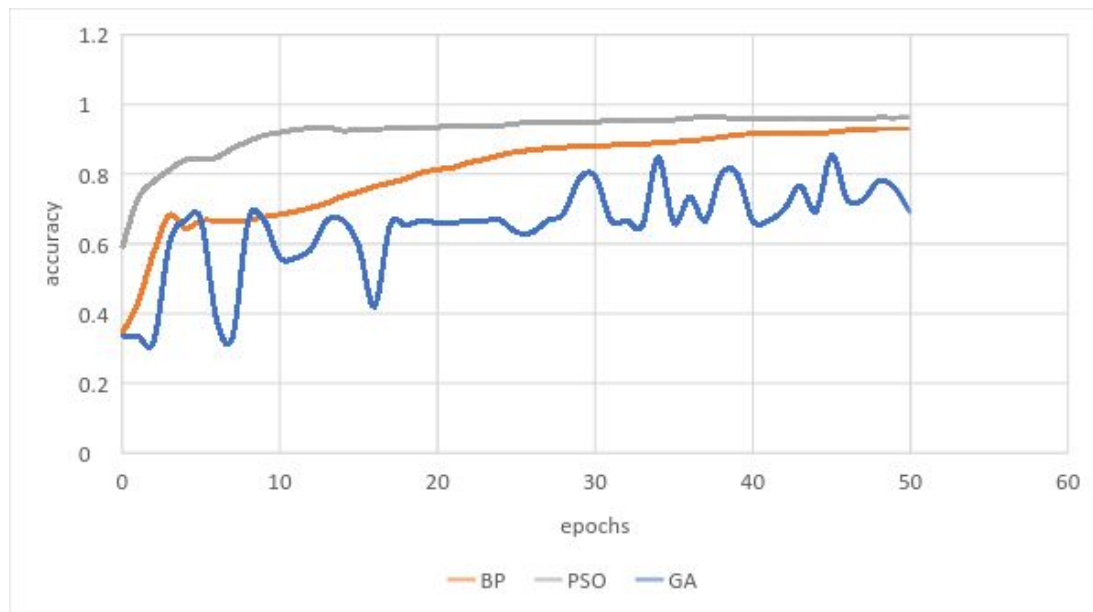
## Iris Data Set

Iris data set is the first subject that we dissect. For the iris data set the average mean squared error for each technique from 0 to 50 epochs is displayed in a graph below. The following chart is constructed by taking averages of the data set from 10 different runs of the program.



We can see some fascinating results that were not expected. PSO and GA are acting in the same manner as the hypothesis. However, BP is underperforming. BP's error at the start is quite significantly low compared to PSO and also converges on the solution much slower. Meanwhile, GA is very different from both BP and PSO where it has more of a straight line going down rather than quickly getting the smaller error than slowing down. GA has the same speed slowly converging on the error. However, let's not draw premature conclusions from one graph and see the next set of results.

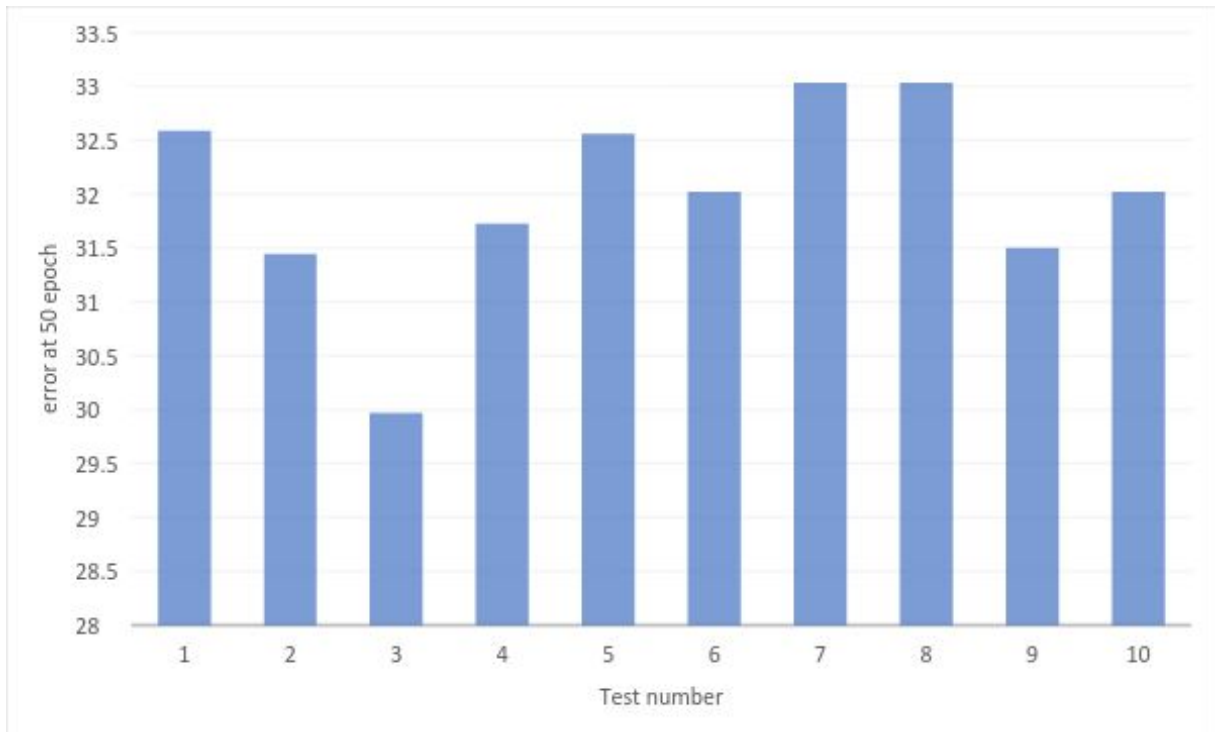




The above graph shows how accuracy is improved over each epoch. Looking the chart, we can draw our attention to GA which acts a bit suspiciously where the accuracy is fluctuating massively. At first glance, it seems quite perplexing and might seem like it is not working properly. However, if we apply some thought, we can deduce that GA is acting in this manner since it uses mean squared error to improve the neural network and sometimes having a lower mean squared error can also have lower accuracy overall. For example, if the desired output is 1, 0, 0 then getting 0.5, 0.49, 0.1 will mean that the neural network got the correct solution, raising the accuracy but the error is massive. Moreover, if we look at the broader picture it is, in fact, reducing the error but just fluctuating while getting there.

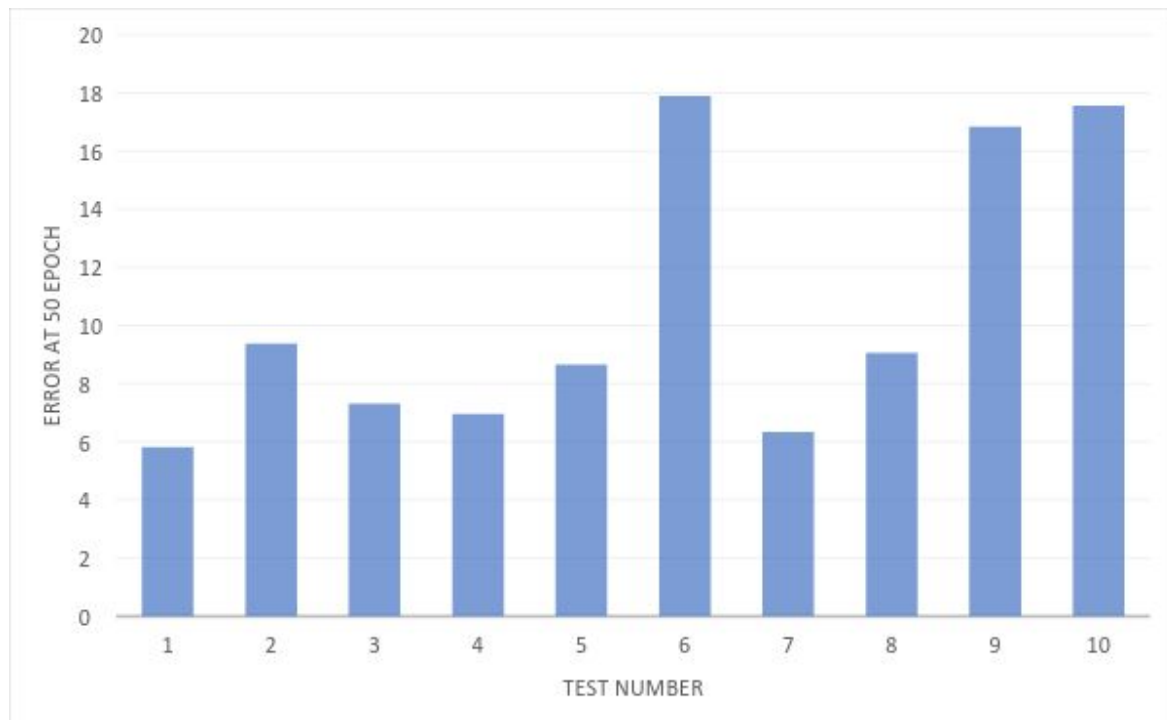
On a different note, we can now see using this graph it seems BP and PSO are almost similar after 50 epoch BP appear to be catching up to PSO but the first graph showed that BP was very behind PSO. This seems to be due to the same principle that error and accuracy are not linked. However, if anything, this graph proved is BP is better regarding accuracy but noting the difference in values we can see using the error BP falls quite behind PSO.

The following graph shows all 10 tests at 50 epochs for BP. This allows us to see if the technique can train the neural network each time just as good or the technique is just using luck of the draw.



The graph shows that the minimum error BP has gotten is in test 3 where the error is only 30 while the highest error is 33. Having the difference between maximum error and minimum error it can achieve by only 3 is a good result for BP. This is a good sign that no matter how many tests are run it will be able to train the neural network leading to an acceptable error and accuracy.

Next up is to check if PSO can repeat the similar score at 50 epochs. So, using the same process we can look at the following graph to draw a conclusion.



It seems as if PSO is good but less repetitive then BP where in 3 iterations of the test the error skyrocketed to above 16. However, on other tests it was below 10. Keep in mind this is still lower then BP's error of 30 at the best.

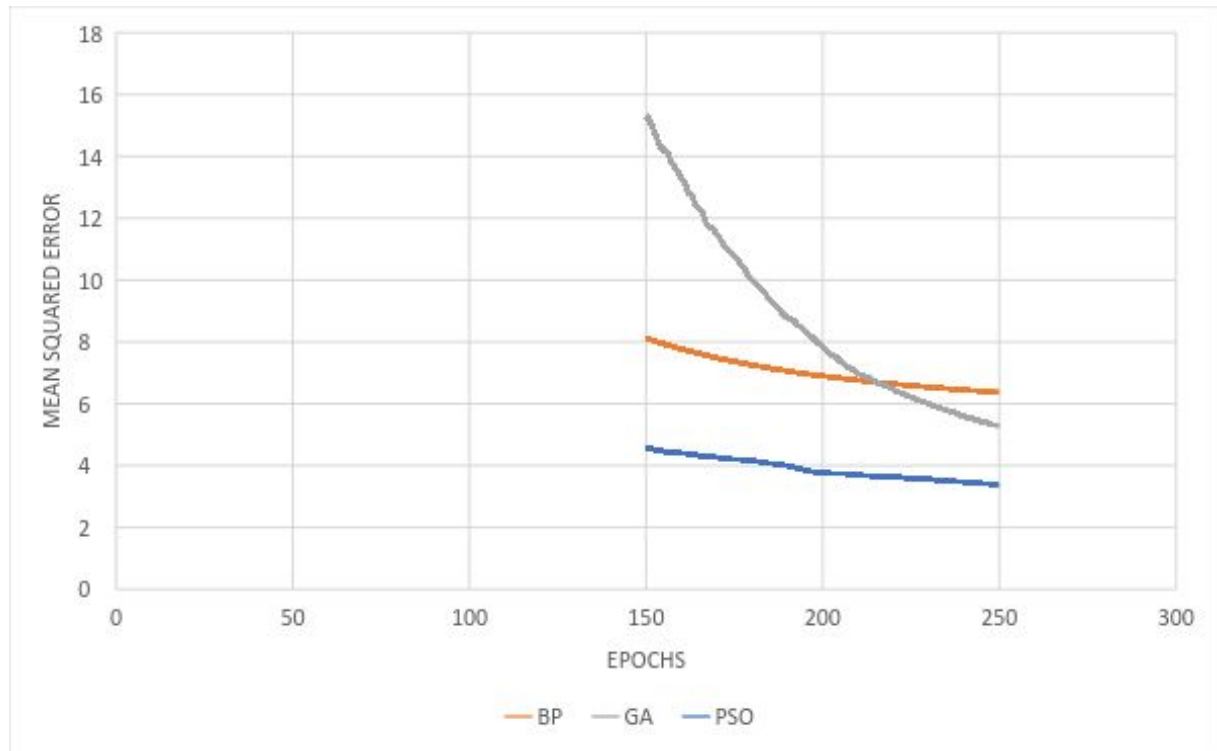
The next technique GA needs to be test and following the accuracy stage if we can expect anything is that the results will be unstable.



The expectation was the minimum and maximum errors would be a lot wider however they are still wider then PSO proving that GA is the most un reliable technique since each

iteration the time taken to train could be massive or quick.

Having a look near 200 epochs range we can see something interesting happen.



As we can see later on BP gets so slow that GA overtakes BP. This graph clearly shows the true beauty of GA as predicted in the hypothesis. GA takes very long time to compute but gets better as time goes on. On the other hand, the other techniques are very quick and fast at computing the weights but they get slower as we go into higher epochs. Keep in mind these tests were running for 1k epochs and in that time no iteration of GA was able to get ahead of PSO for this data set.

This doesn't look all too great for BP. It is getting slaughtered by the other techniques. But the last set of results will tip in BP's favour. This includes the time required to process each epoch. Keep in mind this can vary depending on the processing power.

Average Time Per Epoch in microseconds

BP: 3184

PSO: 29958

GA: 29550

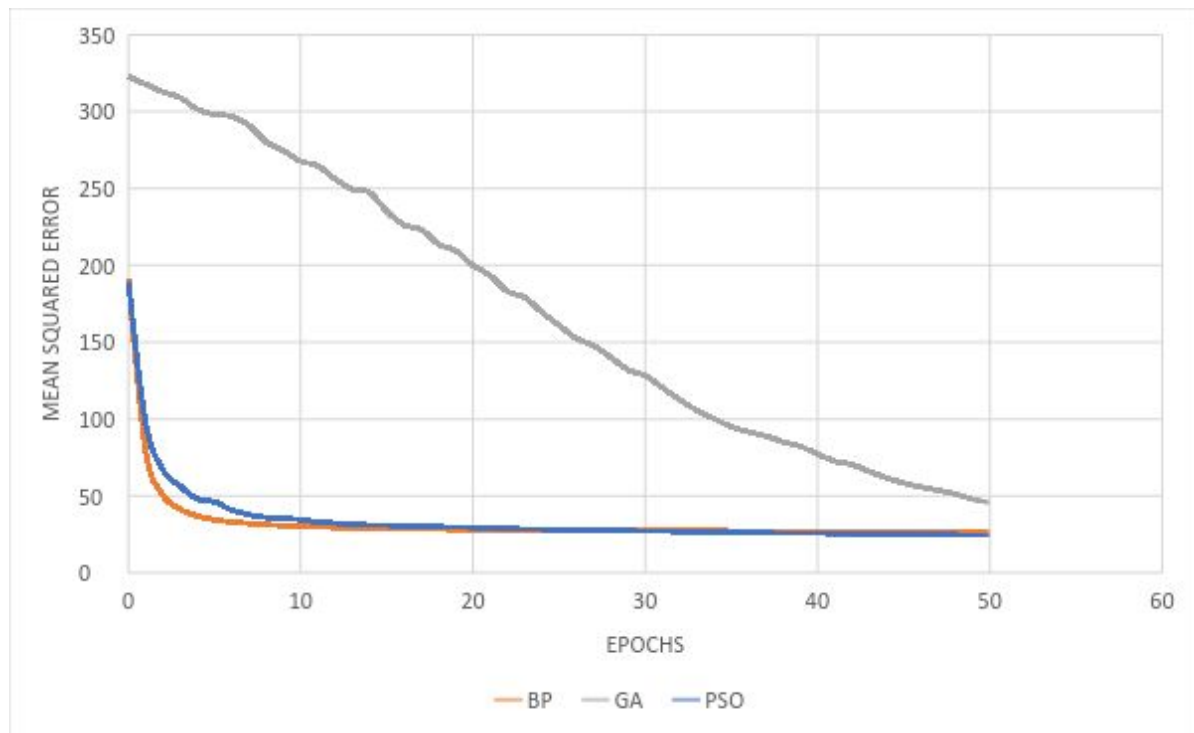
The above values indicate that BP has a massive lead in time taken to compute 1 epoch which could mean that BP can do ~470 epochs in the time PSO and GA will take to do ~50. Taking this into account BP has a massive lead over PSO and GA. In ~1.5 seconds of training BP's error is ~5.73 but PSO's error is ~10.58. Meanwhile GA is left in the dust with an error of ~57.3 at ~1.5 seconds of training.

Although BP seems to be working extremely well one thing to note here is that near ~5.6

error BP seems to slow down to a halt not reducing the error in over 200 epochs. While GA and PSO both seem to keep lowering the error as the program continues on further. From this we can gather that where high accuracy is required without any time restrictions PSO is the way to go but if you have time restrictions BP is far superior while GA is lagging behind the both techniques.

### Breast Cancer

Using the same process as iris data set, breast cancer data set is analysed. Firstly, looking at the 0 to 50 epochs averages of the 10 results of each technique.

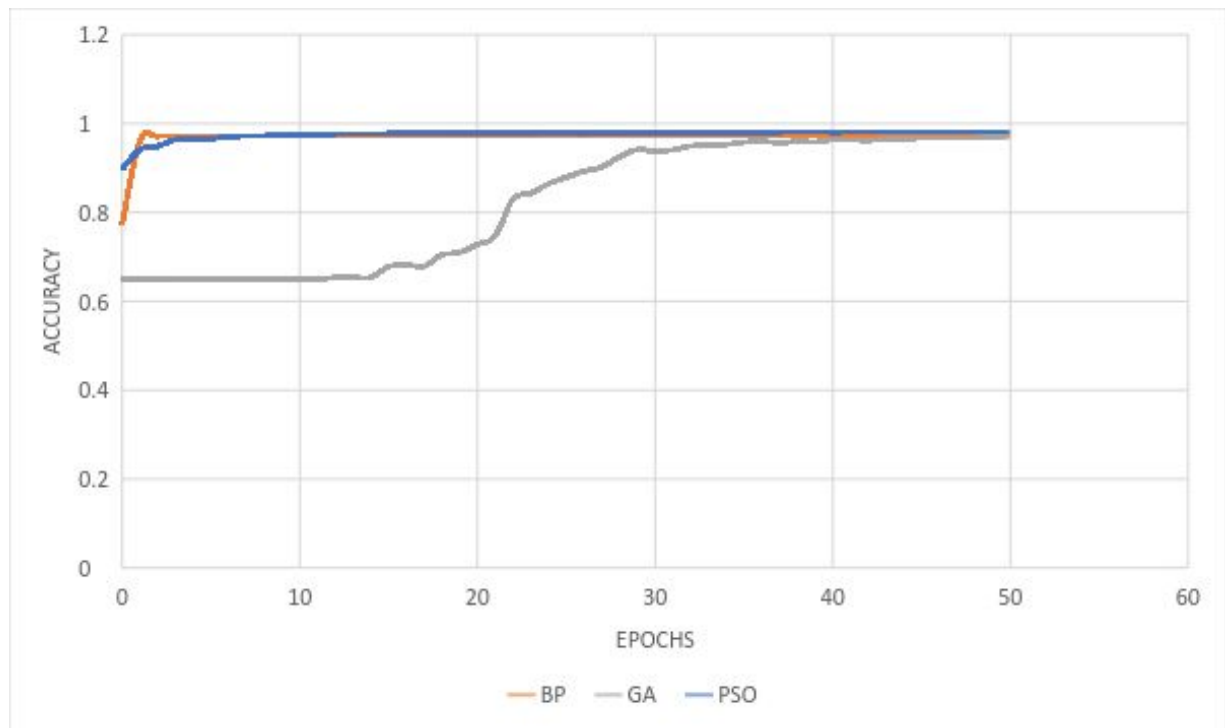


This graph shows some interesting results, contrary to the iris data results, in breast cancer BP is ahead, if not the winning near 5 epochs. Why is this happening? To find the answer we can first look at what has changed.

From the Iris data set only 2 things have changed. The neural network construction itself and the training data since we are now using breast cancer data set. The only difference found in the training data from iris is that breast cancer uses whole numbers meanwhile the iris data set uses real numbers having decimal points. This could have an effect on the training process. The other thing is that in breast cancer a neural network with higher amount of weights and bias is being used to train the neural network. This is because the neural network itself has more input, hidden and output nodes. This can translate to BP being better at training larger networks. However, drawing that from one graph is preposterous and these 2 changes will be explored further in this report.

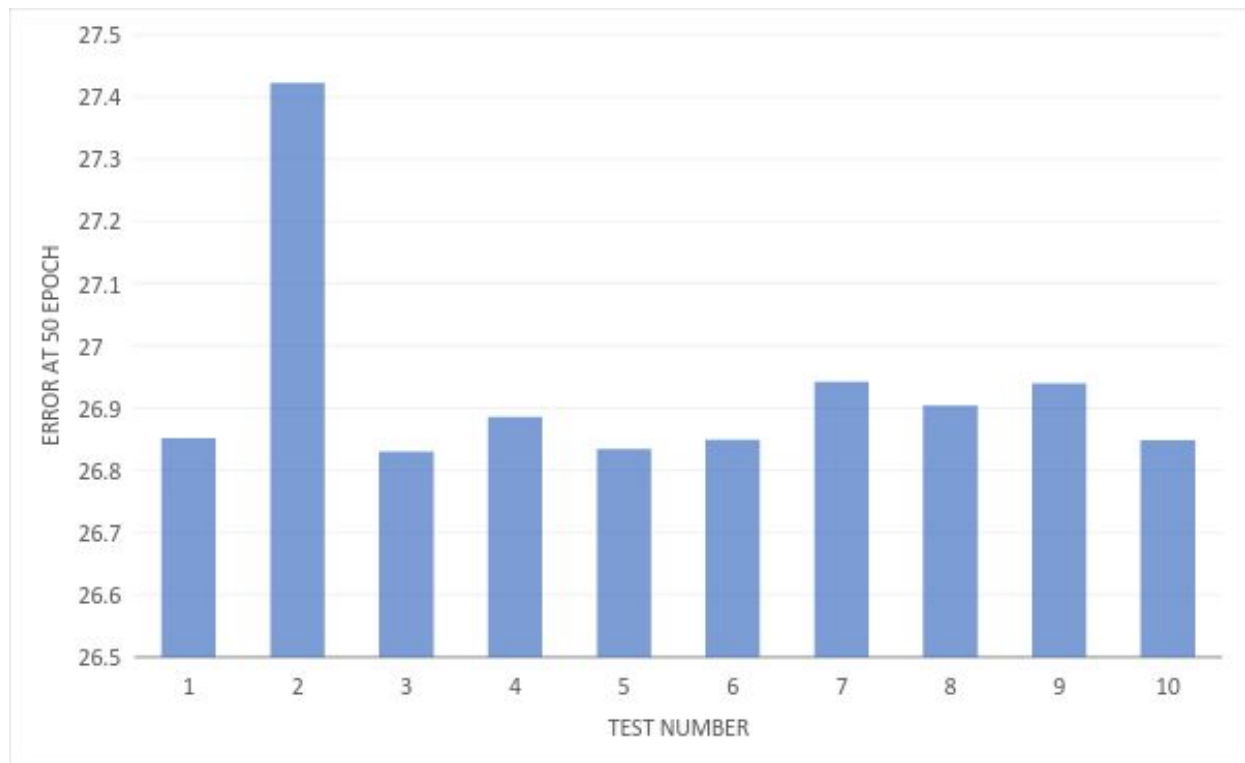
After the mean squared results the accuracy is expected to be BP's favour and looking at the following chart it indeed shows that. But we can see that PSO is a close second while GA takes very long but eventually do catch up to both techniques. Interestingly in the mean

squared error PSO does overtake BP and in the accuracy, we can see the same result. It is a minor difference but still there.



One of the interesting things to note on the accuracy graph is the very beginning. We can see that PSO is in fact doing a lot better at the first few epochs. But BP overtakes PSO by increasing its accuracy by a massive amount. With further analysis, we can draw the conclusion that BP is very good at quickly converging on the correct solution by shows poor results after it has been running for a while. Meanwhile PSO's convergence on the error is good but below BP's standards. The up side of using PSO is that from the current results it seems it will always take over BP in the long run. Looking at the GA it seems as it is a lost cause since it is taking far too long comparing it to the other techniques.

In the next set of tests, we will look at the repeatability of the techniques to produce a sufficient error at 50 epochs.



BP shows less than 1 error difference between the lowest and the highest error recorded in all the tests done. This is a big win for BP over the other techniques and BP is looking to be the superior technique overall.

The next graph shows the same error at 50 epochs for PSO in 10 different iterations of the test.



The PSO's results show quite a bigger difference between the minimum and the maximum

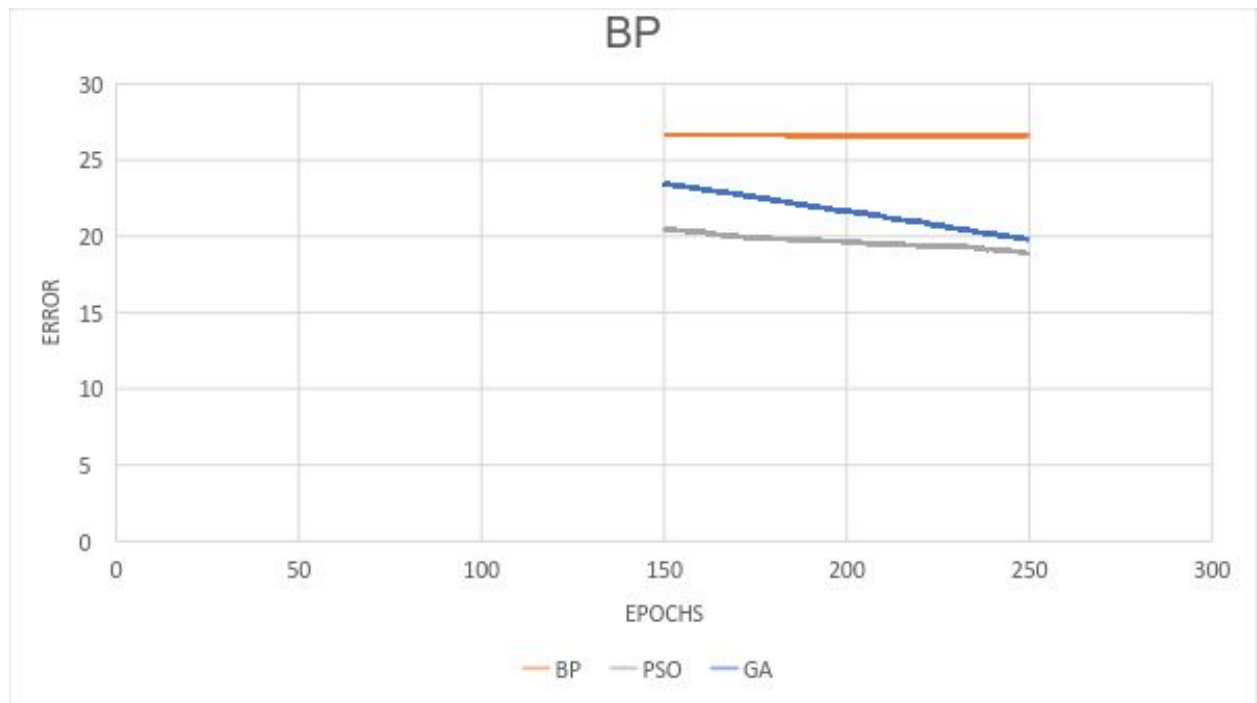
error record across 10 tests. In some cases, PSO's error is actually worst then BP. Again, showing the BP to be winning in this comparison. Lastly, we check the GA graph expecting it to be far worse than other techniques.



Indeed, our expectations were true. Right now, it seems as GA is a lost cause showing incredibly bad performance in training the neural network.

Now we can look at the techniques further along in the training process. The following graph shows the techniques error at near 200 epochs.





In this graph, the roles have reversed. As we saw previously BP was winning by a mile but now BP is the slowest technique with the error at a complete halt. In this we can see even the slowest technique GA has overtaken BP. While PSO is still in the lead and GA seems to be closing up the distance on even PSO. After the first few epochs BP seems very weak meanwhile PSO and GA both are decreasing the error more and more.

Lastly just like the iris data set we have to look at the time in microseconds required to complete each epoch on average for each technique.

BP: 14355

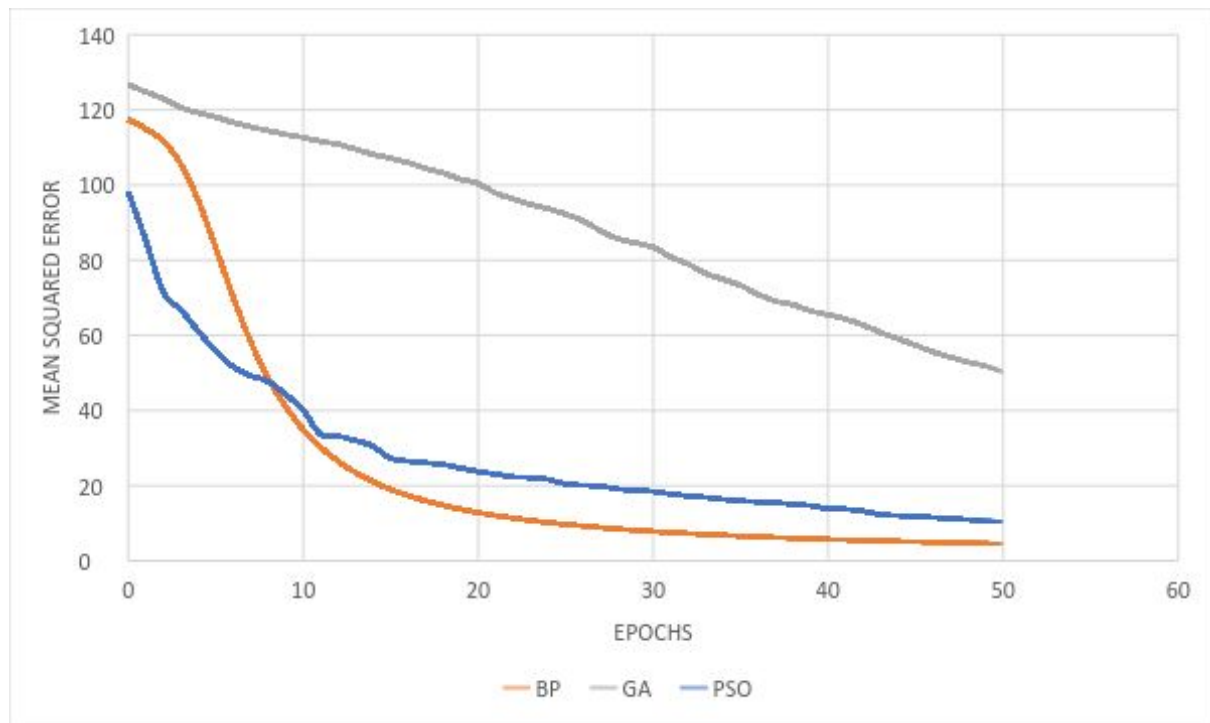
PSO: 126521

GA: 127603

Again, just like the iris data set BP takes the win here but this is to be expected. More importantly now the real question is. Is that extra time useful for BP? The previous graph shows that after the first few epochs it does not minimize the error at all. So, if the user is looking for more accurate neural network the other techniques (PSO & GA) might be better option.

## Wine

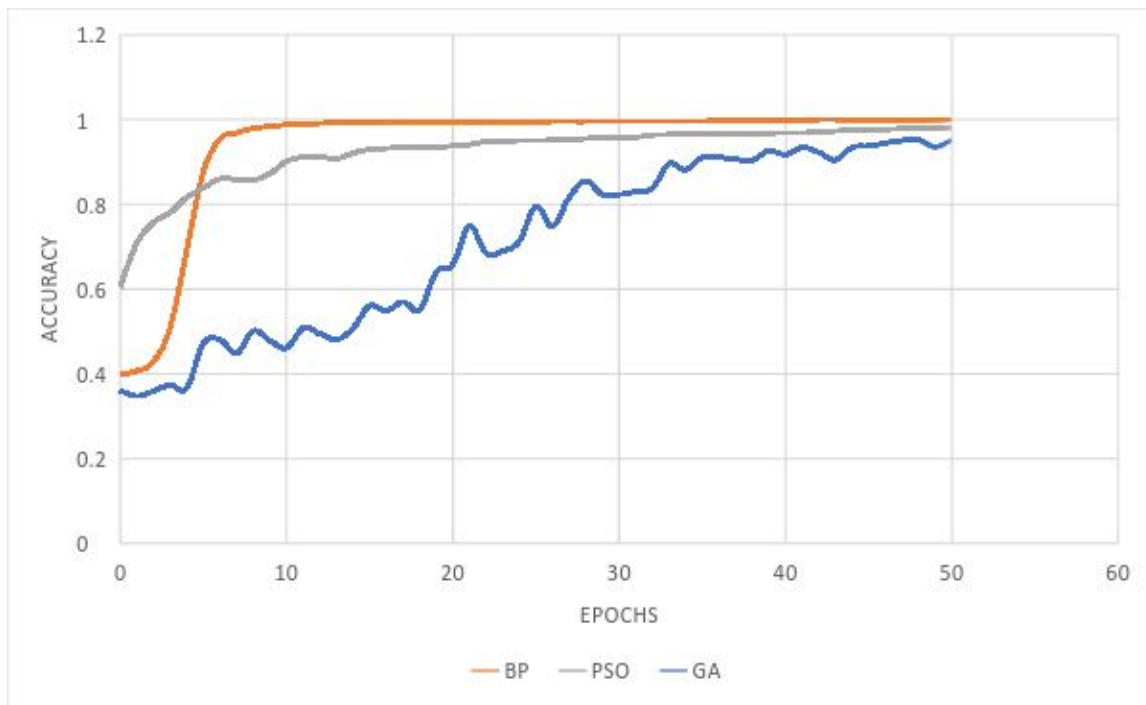
After seeing the very different results of breast cancer data set it was assured that more testing needed to be done. The next data set to go on the chopping board is wine data set. Using the same methodology, the following graph shows the first 50 epochs mean squared error for each technique training the wine data set.



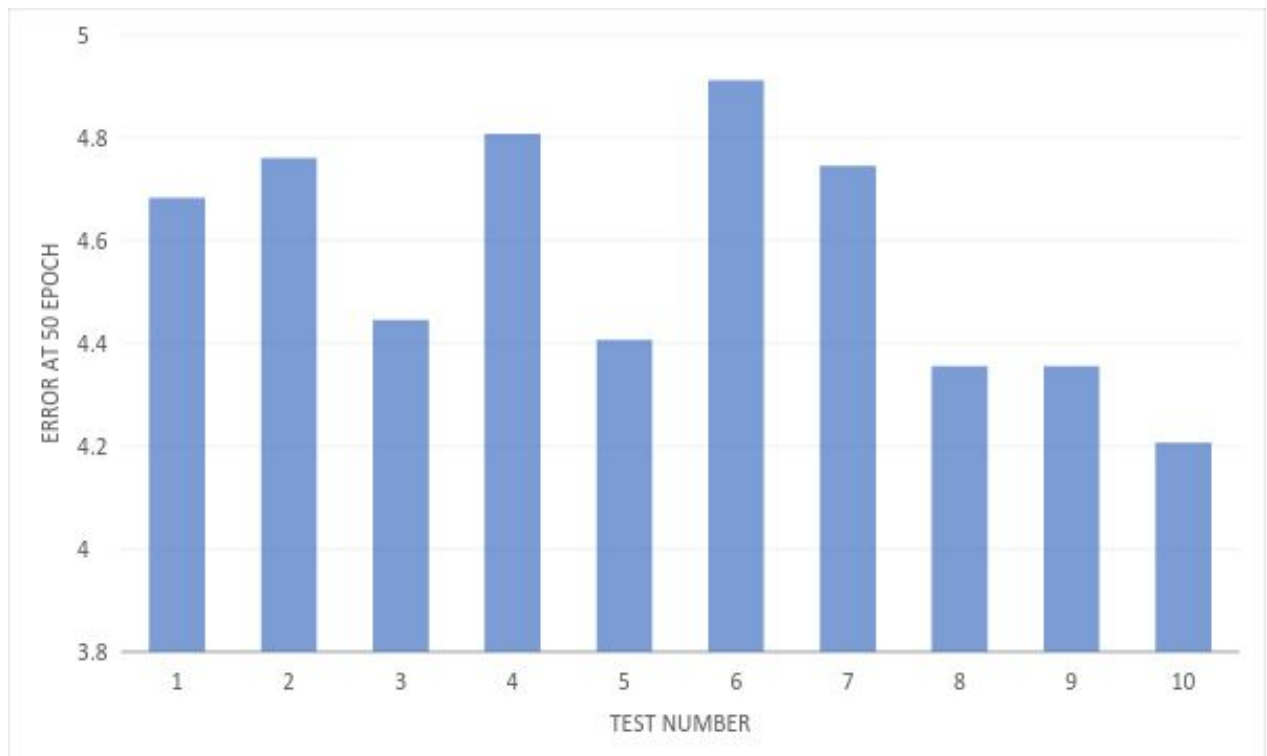
Using all the previous results and this graph a clearer picture is now formed. It seems that PSO usually starts with a lower error in first few epochs. While BP has a higher error in the first few epochs. This quickly changes because BP seems to be very effective at converging to a minimal error; while PSO is slower. However, PSO does appear to take over BP yet again if the training is lasting quite a long amount of time. This is bad for BP as its greatest advantage of speed is now also being knocked at with a hammer by PSO. The above result shows that anyone looking for training a neural network really quickly PSO can get you a minimal error faster than BP even with BP's speed advantage. And if your intention is to run the program with no time restrictions to get the best accuracy then the previous result shows that PSO is better in that regard as well.

GA is in its own league with really slow convergence on the minimal error but very effective at longer period of time as it seems.

Looking at the accuracy it looks the same result from above saying the same thing that PSO has better accuracy. Followed by BP taking over with quicker convergence to the error. However, PSO does catch up and take over. This cannot be seen by this graph since for this particular data set BP reaches 100% accuracy. But the next set of results prove this point further that PSO and GA both are capable of overtaking BP in the long run.



Now checking the repeatability of errors at 50 epochs again for each technique starting off with BP.



As always BP's repeatability is very high showing that each time its trained, the training error will be identical to last iteration of training. Next, we check the repeatability of PSO.

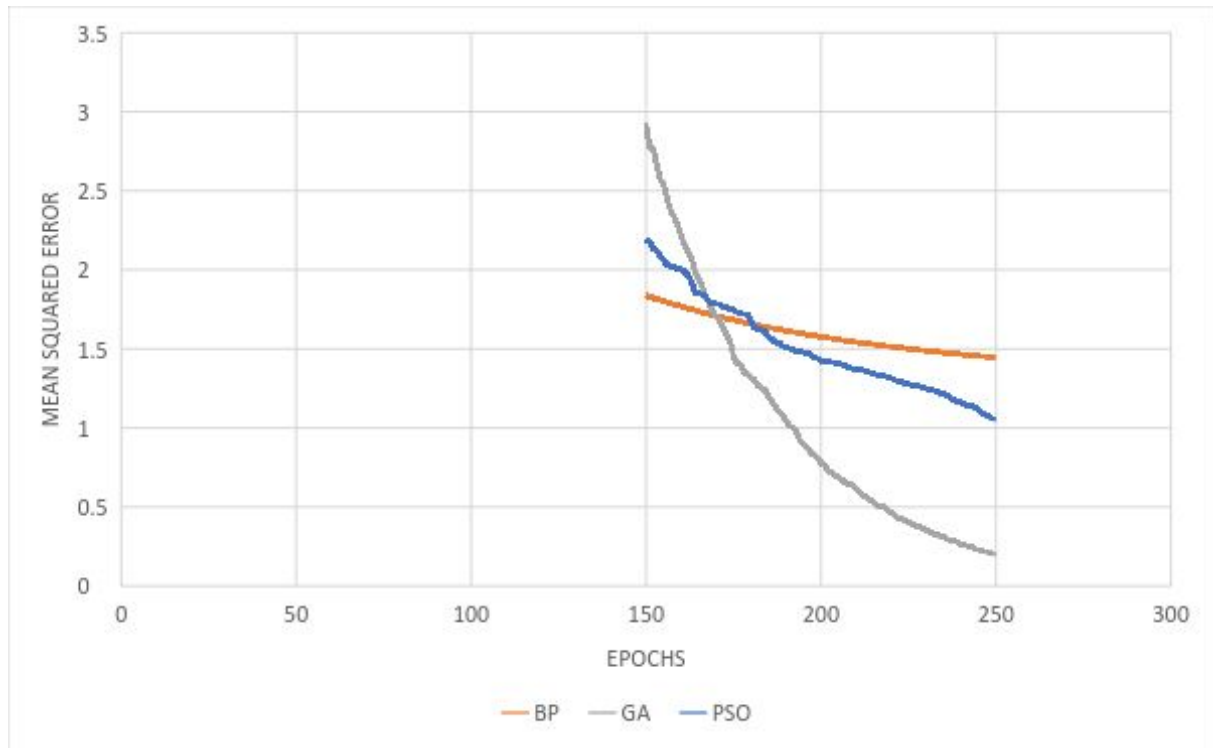


Again, PSO is showing wide disparity that each time you train a neural network the results can vary by considerable amount using PSO. With BP and PSO done now we have to check GA's repeatability, expecting it to be a little worst then PSO.



We see GA's repeatability is very similar to PSO but just a little worst as expected. So, it seems in this front BP is superior if you are planning to train the same neural network multiple times with the intention of having the same error each time you train it. For a set number of epochs.

Eager to see the long run PSO & GA results for this data set first we will look at the data showing the error near 200 epoch range. In the next set of results, we expect PSO & GA to take the lead over BP.



And the results are even better than expected. This graph shows although GA is slow, in the long run it out performs the other techniques. Also, the true flaw of BP is shown here, to an untrained eye BP's convergence on the error seems like it is far better alternative but when the training begins PSO has a lower error and when it ends PSO still has a lower error. BP is only good in the mid-point and quickly falling of the longer the training continues.

Lastly the time required to go through one epoch is checked again for each technique.

BP: 4735

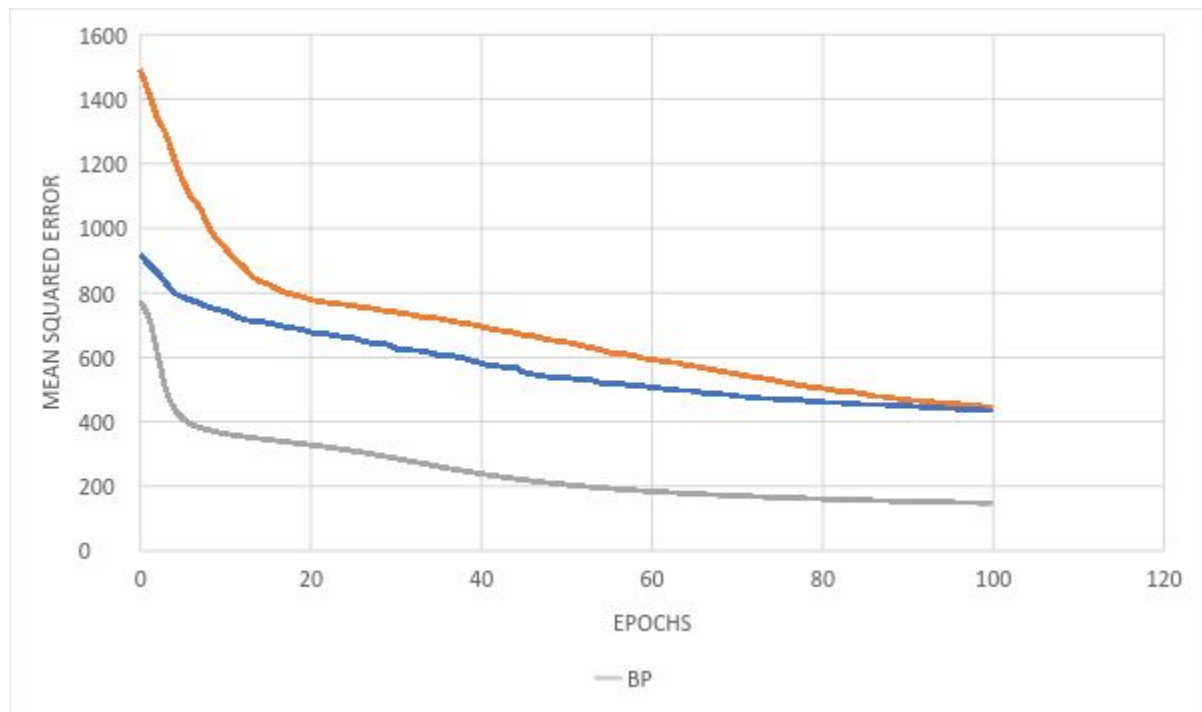
PSO: 38337

GA: 38552

Again, we get the exact same boring result. This is not very interesting as it means in every case BP will be faster per epoch from PSO and GA.

## Car Evaluation

Massive amount of car evaluation data was also used in this experiment. Iris and wine data set contained ~200 items in the training data set, breast cancer has ~700 items in the training data. Meanwhile car evaluation has 1700 items in the training data. For this reason, the mean squared errors are expected to be quite high. Starting the analysis with first looking at the first 100 epochs this time because the mean squared errors are high.



In the above graph, something very different has occurred from all previous test runs. BP is starting ahead of PSO for some reason. To find out the reason why let's look at the 2 things discussed in breast cancer data. The changes occurred from iris to breast cancer and breast cancer to car evaluation.

In iris only 63 weights and bias were used in a neural network containing 4 inputs, 7 hidden and 3 output nodes. For breast cancer 95 weights and bias are used in a neural network of 9 inputs, 7 hidden and 2 output nodes. Finally, for car evaluation data 363 weights and bias were used with a neural network having 21 inputs, 13 hidden and 4 outputs. Looking at the results this can mean that the bigger the neural network the better BP performs.

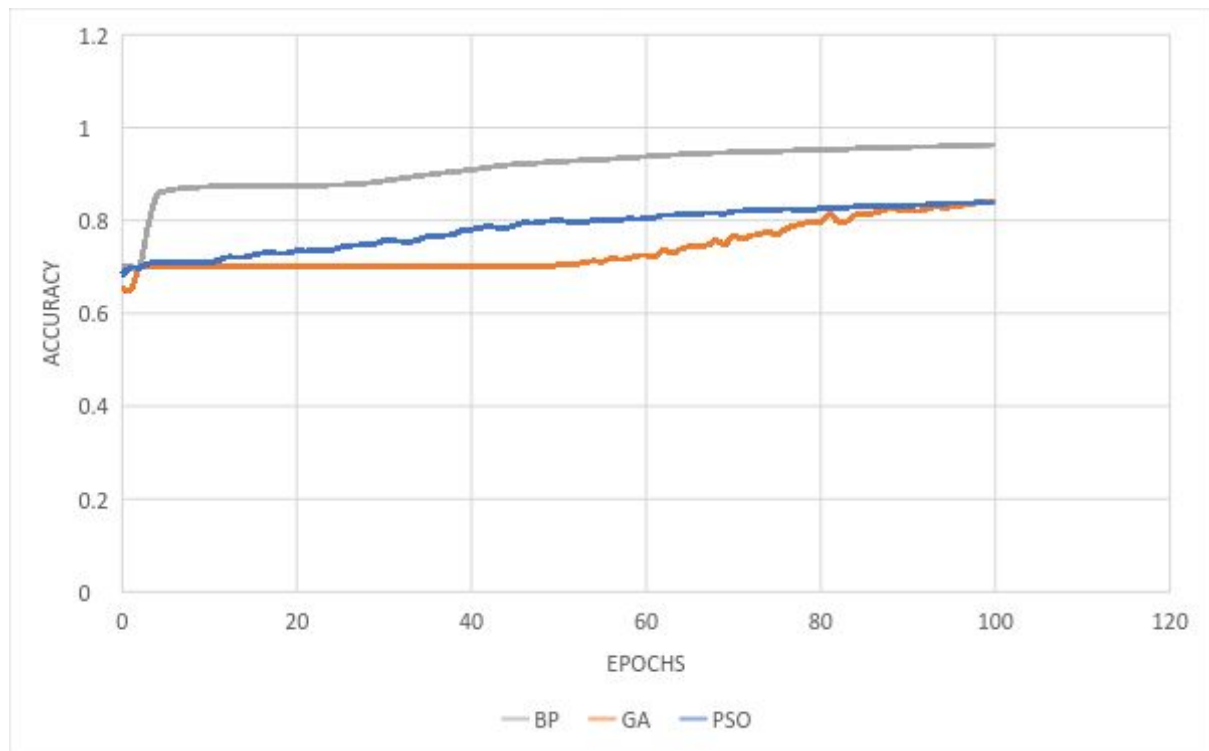
The other thing changed was the training data but increasing the training data size should not accomplish anything more than slowing down the training.

From analysing these 2 subjects it is thought that since the data is so large with the neural network size increase BP has a better advantage. The reason for this is simple if we follow our previous findings that BP converges on the minimum error more quickly. By increasing the size of the neural network and the training data size it seems we are also increasing the training time required to train the neural network. Therefore increasing the time, it takes to converge on the minimal solution so it seems that BP has taken the lead thus far. For this to

be proven true we have to look at the tests near the end of this car evaluation test.

One thing that a sceptic might say is that BP starts with a lower error and that destroys our current findings of what is happening. However, if the convergence to the minimum error is very high than in the first epoch the error can be reduced massively. This can be seen from breast cancer data set where the data size is bigger than iris and wine but smaller than car evaluation. Therefore, BP is starting its error much closer to PSO.

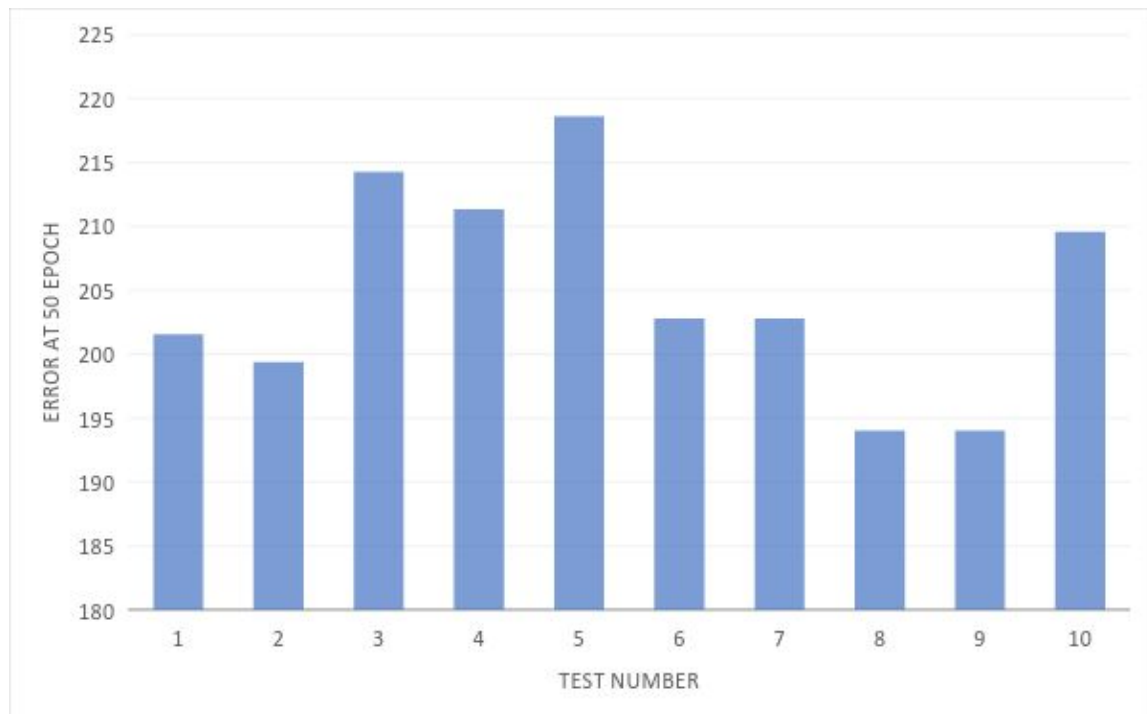
After getting very nice findings for car evaluation test now we must also look at the accuracy data to see how it is effected with massive data and big neural networks.



The results are as expected for BP and PSO but GA seems to be performing quite a lot better in terms of accuracy. Which means it must also get some kind of boost from using large data but it is unlikely that it gets a boost from using large neural networks since only the training time is increased by using large neural networks. This could be effected with GA's mutation and it being able to work better with larger sets of data.

Similar to other results we need to compare how repeatable these results are for larger data sets, starting with BP.

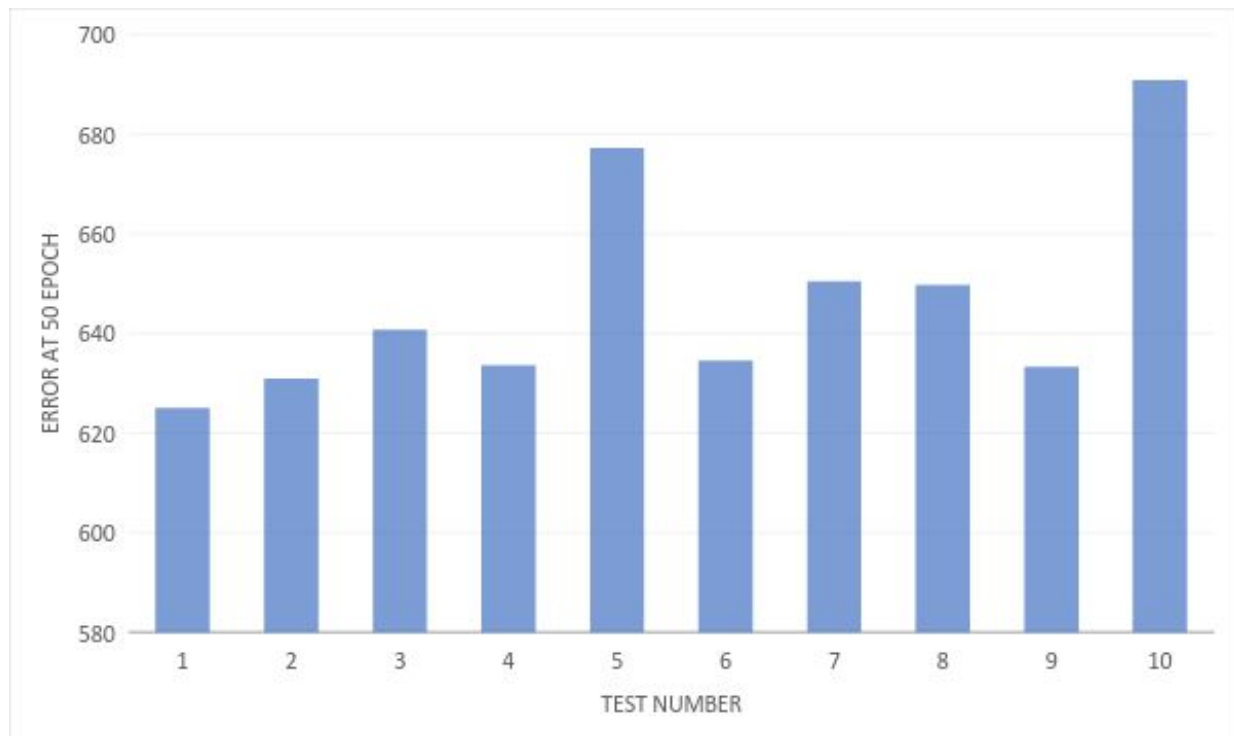




Contrary to what we have seen thus far this time the BP's repeatability score is very low. In test 8 & 9 getting below 195 error while in test 5 BP gets higher than 215 error. That is 20 error difference and in previous results we have seen smaller differences from GA. With this finding it seems more reasonable to show the other results starting with PSO.

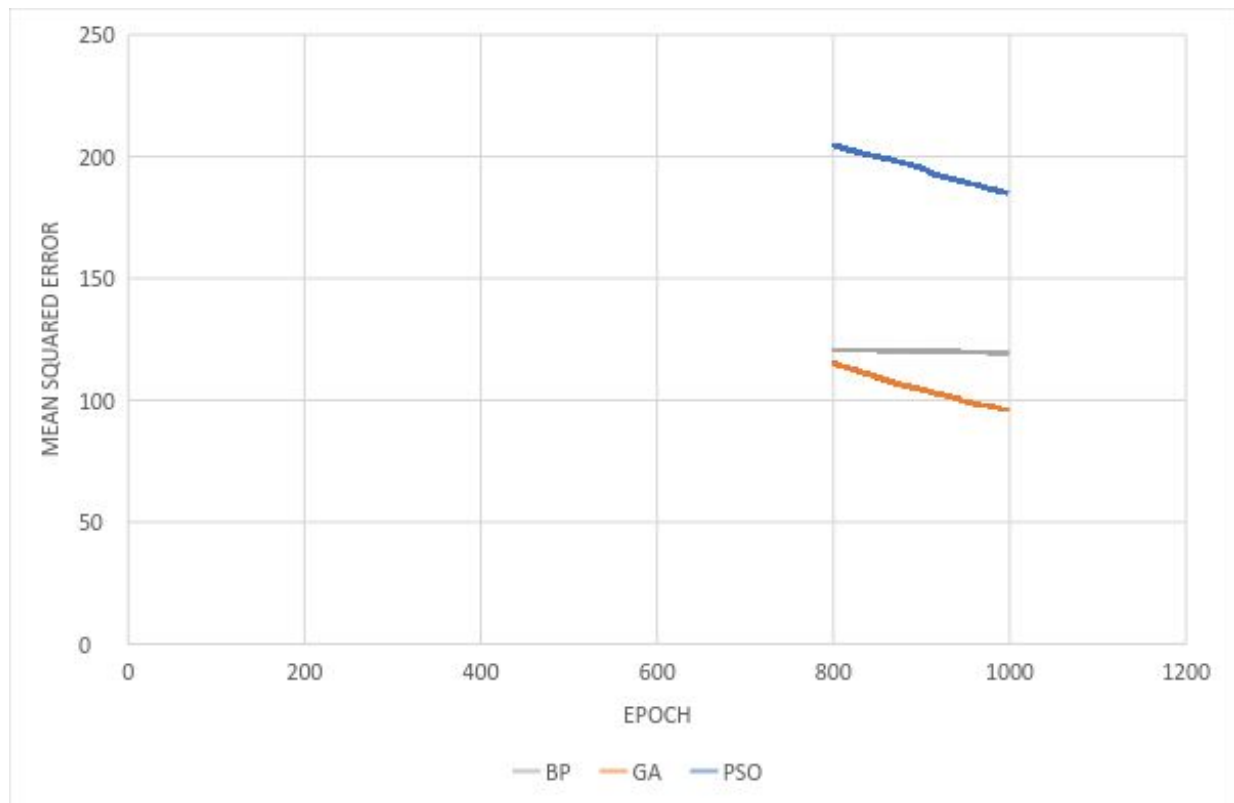


As it seems PSO is now even more unreliable. This indicates that the bigger the training data the more unreliable the technique gets to get the same error. To get a complete picture lets continue on the adventure and look at GA's result.



Just as predicted GA's unreliability has gone through the roof. The test 10 gets near 690 while in test 1 we get 625. Although it seems that PSO is more effected by this than GA.

With the reliability results out of the way lets continue on the end of the testing phase to see if our consensus are correct and the convergence on the error is just lasts more epochs with increased training data and neural network size.



The results might look like that we were wrong but upon further inspection we can see BP is not lowering the error anymore and indeed PSO will overtake BP since its still declining but unfortunately only 1k epochs were ran during the testing phase. But from past examples and with GA already being below BP; it is safe to assume that PSO & GA will get lower error as time goes on.

Finally ending these results with a final duration test on how long each epoch took on average to compute.

BP: 90061

PSO: 663734

GA: 659204

This result is as expected where BP wins by a long shot. Meanwhile PSO and GA are completing each epoch near the same amount of time. However, one thing to note is each techniques epoch per time has a direct relationship with the training data size. The bigger the training data, the longer the time required per epoch to train.

## 6. Future Work

This is groundwork for future neural network training research. In this project only a basic MFNN was used to train and show that BP should not be used in every situation but in many cases the other techniques could be better suited to the task. Furthermore, different structures of neural networks will have different effects on the training with each technique performing differently. This also can be a topic on further research as the limitation imposed

on this project will not allow such massive research to be accomplished in such a short time.

## **7. Conclusion**

In conclusion, the results from our analysis show that for large neural networks it seems that BP is a good starting off point but to get the full advantage GA or PSO should also be implemented because BP always seem to slow down to a halt after a few epochs. While PSO and GA both techniques seem to overtake BP in the long run. GA is better at longer period of time than PSO, However PSO starts with a lower error.

If using smaller neural networks than PSO takes the cake. Its error is far lower after just a few epochs and is still ahead in longer period of time, while BP again stagnating to an error. In addition to that BP also starts at a much higher error for smaller neural networks than PSO. The PSO technique is a very good balance between BP and GA if the aim is to have a technique good at converging at the error without losing longevity.

GA is by far the weakest technique really early on but if you give it time It improves by a drastic amount. If the user has infinite time than GA is the way to go since it is guaranteed that GA will overtake both techniques in the long run. GA's power also increased with the size of neural network. We can see that from all the results as we increase the neural network size, GA performs better and better in the long run. Looking at the car evaluation test GA overtakes BP and PSO both techniques after ~600 epochs.

## 8. References

- Abbas, Q., Ahmad, J. and Bangyal, W. H. (2013) 'Dynamic Hidden Layers Selection of ANN Architecture Using Particle Swarm Optimization', *International Journal of Engineering and Technology*, 5(2), pp. 195–197. doi: 10.7763/IJET. 2013.V5.540.
- Cireşan, D. C. and Schmidhuber, J. (2013) 'Multi-Column Deep Neural Networks for Offline Handwritten Chinese Character Classification', *arXiv preprint arXiv:1309.0261*, p. 5. doi: 10.1109/IJCNN.2015.7280516.
- Forrest, S. (1993) 'Genetic algorithms: principles of natural selection applied to computation.', *Science (New York, N.Y.)*, 261(5123), pp. 872–878. doi: 10.1126/science.8346439.
- Giesl, J., Esponda, F. and Forrest, S. (2001) 'Genetic Algorithms for Finding Polynomial Orderings', *Citeseer*, (0), pp. 1–18. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.75.3638&rep=rep1&type=pdf>.
- Forrest, S., Javornik, B., Smith, R. E. and Perelson, A. S. (1993) *Using Genetic Algorithms to Explore Pattern Recognition in the Immune System, Evolutionary Computation*. doi: 10.1162/evco.1993.1.3.191.
- Gupta, N. (2012) 'GENETIC ALGORITHMS: A PROBLEM SOLVING APPROACH What is Genetic Algorithm?', (1975), pp. 940–944.
- Kennedy, J. and Eberhart, R. (2002) 'Particle swarm optimization', *Proceedings of IEEE International Conference on Neural Networks*, 4, pp. 1942–1948. doi: 10.1109/ICNN.1995.488968.
- Kennedy, J. and Eberhart, R. (1995) 'Particle swarm optimization', *Proceedings of ICNN'95 - International Conference on Neural Networks*, pp. 1942–1948. doi: 10.1109/ICNN.1995.488968.
- Mahajan, R. and Kaur, G. (2013) 'Neural Networks using Genetic Algorithms', *International Journal of Computer Applications*, 77(14), pp. 975–8887. doi: 10.5120/13549-1153.
- Maind, S. B. and Wankar, P. (2014) 'Research Paper on Basic of Artificial Neural Network', *International Journal on Recent and Innovation Trends in Computing and Communication*, 2(1), pp. 96–100.
- McCulloch, W. S. and Pitts, W. (1943) 'A logical calculus of the ideas immanent in nervous activity', *The Bulletin of Mathematical Biophysics*, 5(4), pp. 115–133. doi: 10.1007/BF02478259.
- Mhaskar, H. N. and Micchelli, C. a (1994) 'How to Choose an Activation Function', *Advances in Neural Information Processing Systems* 6, pp. 319–326. Available at: [http://papers.nips.cc/paper/874-how-to-choose-an-activation-function.pdf%5Cnfiles/2354/Mhaskar?Micchelli-1994-How to Choose an Activation Function.pdf%5Cnfiles/2355/874-how-to-choose-an-activation-function.html](http://papers.nips.cc/paper/874-how-to-choose-an-activation-function.pdf%5Cnfiles/2354/Mhaskar?Micchelli-1994-How%20to%20Choose%20an%20Activation%20Function.pdf%5Cnfiles/2355/874-how-to-choose-an-activation-function.html)

- Mitchell, M. and Forrest, S. (1993) 'Genetic Algorithms and Artificial Life', *Optimization*, pp. 1–28.
- McCaffrey, J. (2013) *Neural Network Demo with C#*. Available at: <https://gist.github.com/atifaziz/9462430> (Accessed: 10 February 2017).
- McCaffrey, J. (1996) *Particle swarm optimization using C# -- visual studio magazine*. Available at: <https://visualstudiomagazine.com/articles/2013/11/01/particle-swarm-optimization.aspx> (Accessed: 10 February 2017).
- McCaffrey, J. (2013) *Developing neural networks using visual studio*. Available at: <https://channel9.msdn.com/Events/Build/2013/2-401> (Accessed: 10 February 2017).
- McCaffrey, J. (2015) *James McCaffrey: Swarm intelligence optimization using python*. Available at: [https://www.youtube.com/watch?v=bVDX\\_UwthZI](https://www.youtube.com/watch?v=bVDX_UwthZI) (Accessed: 10 February 2017).
- Rana, S., Jasola, S. and Kumar, R. (2011) 'A review on particle swarm optimization algorithms and their applications to data clustering', *Artificial Intelligence Review*, 35(3), pp. 211–222. doi: 10.1007/s10462-010-9191-9.
- Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (2013) 'Learning Internal Representations by Error Propagation', *Readings in Cognitive Science: A Perspective from Psychology and Artificial Intelligence*, pp. 399–421. doi: 10.1016/B978-1-4832-1446-7.50035-2.
- Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986) 'Learning representations by back-propagation errors', *Nature*, pp. 323, 533–536.
- Schmidhuber, J. (2015) 'Deep Learning in neural networks: An overview', *Neural Networks*, 61, pp. 85–117. doi: 10.1016/j.neunet.2014.09.003.
- Turing, A. M. (1950) 'Computing Machinery and Intelligence', *Mind*, 49, pp. 433–460. doi: [http://dx.doi.org/10.1007/978-1-4020-6710-5\\_3](http://dx.doi.org/10.1007/978-1-4020-6710-5_3).
- Wani, M. (2014) 'Comparative Study of High Speed Back- Propagation Learning Algorithms', *I.J. Modern Education and Computer Science Modern Education and Computer Science*, 12(12), pp. 34–40. doi: 10.5815/ijmecs.2014.12.05.
- Werbos, P. J. (1990) 'Backpropagation Through Time: What It Does and How to Do It', *Proceedings of the IEEE*, 78(10), pp. 1550–1560. doi: 10.1109/5.58337.
- Werbos, P. J. (1990) 'Neural Networks That Actually Work in Diagnostics, Prediction & Control' Available at: [http://ewh.ieee.org/cmte/cis/mtsc/ieeecis/Tutorial\\_IJCNN04\\_Werbos.pdf](http://ewh.ieee.org/cmte/cis/mtsc/ieeecis/Tutorial_IJCNN04_Werbos.pdf)
- Davis, L. (1991) 'Handbook of Genetic Algorithms', *Computer*, II, pp. 1–6. doi: 10.1.1.87.3586.
- Millonas, M. M. (1994) 'Swarms, Phase Transitions, and Collective Intelligence', *Santa Fe Institute Studies in the Sciences of Complexity-Proceedings Volume-*, p. 30. doi:

citeulike-article-id:5290209.

Aizenberg, I. and Moraga, C. (2007) 'Multilayer feedforward neural network based on multi-valued neurons (MLMVN) and a backpropagation learning algorithm', *Soft Computing*, 11(2), pp. 169–183. doi: 10.1007/s00500-006-0075-5.

Tang, Y. and Salakhutdinov, R. (2013) 'Learning Stochastic Feedforward Neural Networks', *Nips*, 2, pp. 530–538. doi: 10.1.1.63.1777.

Lavrenko, V. (2017). Backpropagation: how it works. [online] YouTube. Available at: <https://www.youtube.com/watch?v=An5z8lR8asY> [Accessed 11 Apr. 2017].

## 9. Appendix

<https://github.com/zeeshan595/Neural-Network>