G52GRP Final Group Report

Neural Network Toolbox - Alfred

Group ID: gp14-jl-rij ***Date*****:** Mar.26.2015

Members:

Barnabas Forgo---bxf03u
Hee Kah Ooi---hko04u
James Michael Ronayne---jmr03u
Luke Bates---lxb03u
Mark Allen---mxa03u
Mohammed Ali---mxa23u
Weixuan Tao---wxt24u

Supervisor: Robert John

Table of Contents

Chapter 1: Introduction

Introduction to Neural Network

Artificial Neural networks (ANN) are made up of nodes operating in layers. These nodes are inspired by the human nervous system (Beale, Demuth and Hagan 2008). ANN makes an attempt to simulate the human brain. As for the human brain, it is believed that nodes known as neurons, which are connected to each other through axons and the receptive lines are called dendrites. Creation of new connections and the modification of existing connections are two important mechanisms for the human brain. For a ANN the inputs of the neurons are set out as a layer of different weights. The result of this combination is then fed into a non-linear activation unit (activation function). A neural network can be trained to perform a particular function by adjusting the weights of the connections between nodes. Neural networks are trained, so that a particular input leads to a specific output. The network is then adjusted based on a comparison of the output and the target until the network output matches the target. Here is a picture according to Maltarollo et al(2013)'s research showing the comparison between a human neuron and an ANN neuron:

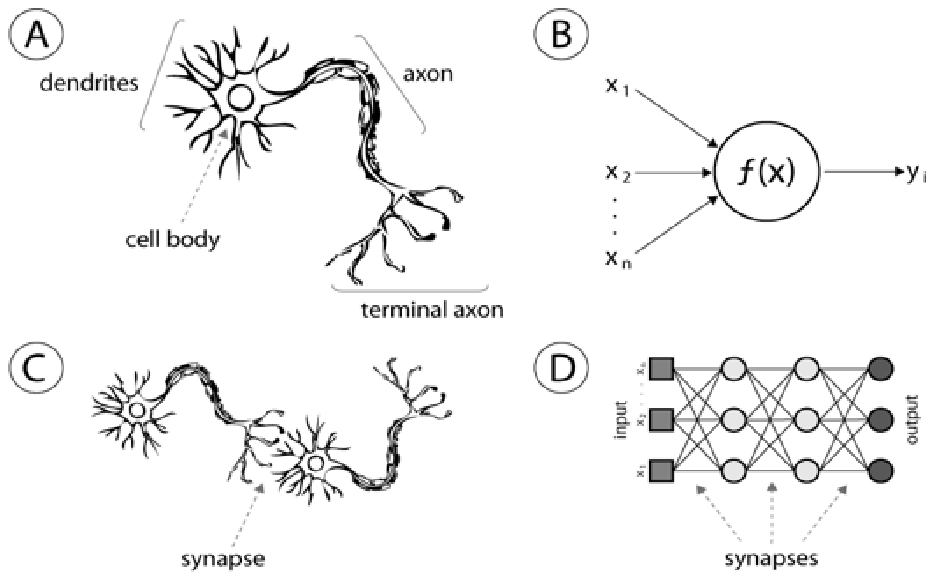


Figure 1.1- *A) Human neuron; (B) artificial neuron or hidden unity; (C) biological synapse; (D) ANN synapses.*

In addition, cited from Fauske(2006) there is a more clear neural network photo. See Figure 1.2:

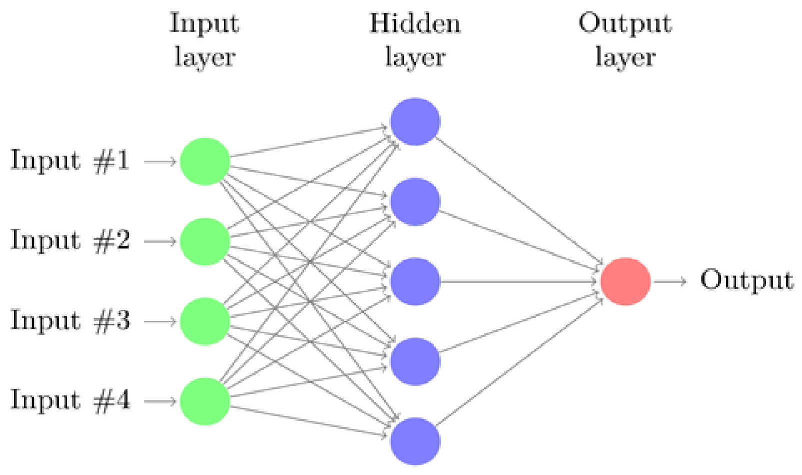


Figure 1.2 *- A typical neural network*

In machine learning, artificial neural networks are computational models that allow for automatic classification and pattern recognition when given a set of data. Neural networks are used to estimate or approximate functions that depend on a large number of inputs. A good example would be credit scoring for a bank where they want a system that decides whether to give someone a loan. So they get a lot of data about previous customers (age, salary, etc...) and whether they paid the loan off and the system learns from that data.

In real-life artificial neural networks are used across a variety of fields including decision making, where decisions can be made based off of previous data that can help better inform a decision. Pattern recognition is another field that widely employs neural networks from face recognition to radar systems where certain planes can be identified from their radar signatures. Neural networks have also been used to diagnose several types of cancer as specific models can be constructed from a large group of patients, which can then be used to better diagnose another patient and increasing their chance of survival with better accuracy.

Product Description

Although there exist a few different neural network toolboxes such as Matlab's and Oracle's, our product will be a concise version of a traditional toolbox, providing basic functions such as creating a network, training a network and providing results such as graphs, based on the trained network. With the toolbox, users can design, train, visualize and simulate neural networks. Users can use the neural network toolbox for a variety of applications such as data fitting, pattern recognition, data mining and many other use cases. There will be a number of different algorithms to choose from, such as backpropagation and Self-Organising Maps, ART-1, and Radial basis function. This allows users to make their models more accurate depending on the type of data set and its size, as some algorithms may be faster than others for larger data sets or more accurate.

As a user of the system he/she should be able to perform the following tasks:

1. Open the program
2. Enter the data as a CSV or TXT file
3. Choose the input data and output data
4. Choose the neural network properties such as number of neurons, the architecture of the neural network etc.
5. Initialize the weights
6. Train the neural network
7. Validate/Test the network
8. Use the network

After a basic introduction about the ANN and our toolbox's functions, the next chapter will show some related researches from the aspects of how the general Neural Networks actually work, different algorithms, and continuation of general research.

Chapter 2: Research

Our knowledge of neural networks was limited at first so we conducted research to look up the basics of how they work, a brief history of it, and how they are implemented in languages such as JAVA. Our findings from this research are as follows: McCulloch and Pitts first described ANN's in 1943 and named it as a perceptron network. The first applications of ANNs were not very promising and supplied bad results. Hopfield (1982) introduced the concept of nonlinearity between input and output data. This new view of perceptron promoted a good improvement in the ANN results. With the help of Hopfield's study, Werbos (1990) proposed the back-propagation learning algorithm, which helped the popularization of ANN. As for the operation of ANN consists of training and running. Training the network means providing inputs and outputs. As for running the network, it provides the input which generates an output. The output may not always be accurate. Also the output generally depends on the configuration of the network.

Neural network algorithms can be categorised into two sections: supervised and unsupervised. For supervised algorithms the input and output data are given, the neural network is used to find the relationship of the two. The learning rule is defined as y=f(x)+e, where e is the approximate error needing to be minimized, x being the input and y the output. In unsupervised learning, there is no expected output data given. Instead the algorithms run a number of iterations likening the nodes to the input data, hence classifying the nodes.

Backpropagation:

Backpropagation is one of the most used neural network algorithms in the field of artificial neural networks. Backpropagation is a supervised learning algorithm thus, there are two stages in backpropagation: training and testing. A set of training data is needed as input data, neural network will learn the pattern in the training data and apply it to the testing data which will produce the output.

In addition, backpropagation neural network can be divided into two main stages : propagation and weights updating. However, propagation can be further decomposed into feedforward and backpropagation. In feedforward, input pattern is propagated layer by layer through the network, producing an output. Then an error signal is calculated by comparing the output to the expected output, the error signal is known as the output activations. In backpropagation, the output activation is propagated backwards and then is multiplied to each individual node. As for weights updating, the output and input activation are multiplied to get the gradient descent of the weight.A percentage of the gradient is then subtracted from the weight. The ratio affects the speed of learning and is thus called the learning rate. The higher the learning rate the quicker the network will train. However a lower learning rate means higher accuracy.

Self Organizing Map (SOM):

The two primary resources this information was gathered from are Heaton (2007) and Buckland (2004). The Self-Organizing Map algorithm is an unsupervised machine learning technique that aims to compress data into less dimensions. A common example is the classification of colours given as RGB input vectors. The algorithm starts with a series of input nodes. Each node is a vector of a varying number of weights. Further more there is a lattice of nodes which also contain vectors of the same amount of weights as the input nodes. The weights in the lattice nodes are randomized based on the range of weights in the input vectors.

Training/Running of the network begins by selecting an input node at random and then comparing it to every node in the lattice. It makes note of the node that is most similar to the selected input node; the node that is most like the input is labelled as the Best Matching Unit (BMU). Once the BMU has been found, the algorithm calculates the neighbourhood of the BMU. This is calculated by (See figure 2.1):

$$\sigma(t) = \sigma_0 \exp\left(-\frac{t}{\lambda}\right) \qquad t = 1, 2, 3 \ldots$$

Figure 2.1 *- Calculate the neighbourhood radius and decay it over time*

This formula decreases the radius as the algorithm continues in iterations. Any node within the radius is considered to be part of the BMU's neighbourhood. Once all values have been checked to see if they are part of the neighbourhood, the ones that are have their weights adjusted so that they are more like the selected input vector. This adjustment is calculated using this formula (See figure 2.2):

$$W(t+1) = W(t) + \Theta(t)L(t)(V(t) - W(t))$$

Figure 2.2 - *Updating the weights that fall in the radius of the BMU*

Where L is the learning rate, W is the original weight, V the input vector, L a difference between the input and old weight and t representing a time-stamp. Finally Θ represents the influence placed on the node. The further away from the BMU, the less the weight is altered. These steps are then repeated for the chosen number of iterations until eventually the nodes become separated by their values and can be classified.

Radial Basis Function (RBF):

The idea of RBF networks is derived from the idea of function approximation. Like most neural network algorithms RBF makes use of Multi-Layer Perceptron (MLP) networks but it takes a slightly different approach. The main features of this approach are:

1. Two-layer feed-forward networks
2. Hidden layer nodes implement a set of radial basis function
3. Output layer nodes use a linear summation function like MLP
4. Training is divided into two stages: Firstly weights from the input to hidden layer are calculated and then hidden layer to output layer weights are calculated.

RBF networks work by employing a curve fitting approach in a high dimensional space, which is comparable to finding a surface in a hidden space which is the best fit for the training data. They use a single hidden layer consisting of multiple 'basis functions' which transform the input in a non-linear manner to a high dimensional space. The reason for this is problems which are not linearly separable in the input space can be found to be linearly separable in the hidden space. Fig 2.3 shows what a 2 dimensional input transformed to a high dimensional space would look like.

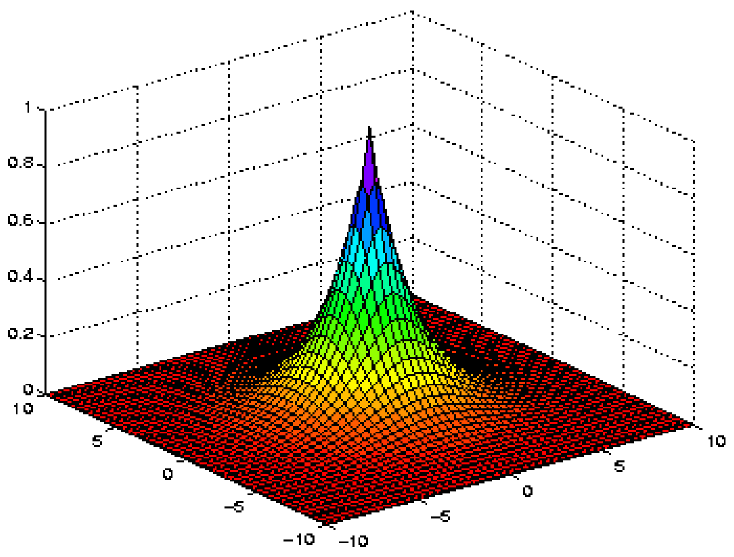


Figure 2.3- *A graphical representation of what a 2d input being transformed to a high dimensional space looks like in a RBF network.*

Both Ringward, Galvin (2002) and Bullinaria (2004) were great sources for further information on how RBF networks are structured and trained.

Adaptive Resonance Theory - 1:

The Adaptive Resonance Theory (ART) network was first developed by Grossberg (1976). The main drive for this was to tackle the plasticity-stability dilemma; namely, how a brain or machine can learn quickly about new objects and events without just as quickly being forced to forget previously learned, but still useful, memories. The way ART solves this problem is that it keeps track of two sets of weights: one top-down from the cluster layer which acts as a long-term memory and one bottom-up from the input layer which acts as a short-term memory.

Additionally ART networks are able to grow new nodes if the newly inputted neurons can't be categorized appropriately with existing nodes. A vigilance factor **v** determines the tolerance of the matching process. A greater value of **v** leads to more, smaller clusters. Figure 2.4 shows a representation of an ART network.

Since ART is a fairly new NN, there aren't many open source implementations so research was a lot harder. After thorough analysis we decided to implement the most basic version of it: ART 1, since other versions were considerably more complex and we wouldn't have had the time to complete it.

One of the major drawbacks of this network type is that it can only work with binary data. In our group after a long discussion we deemed that this was an acceptable limitation.

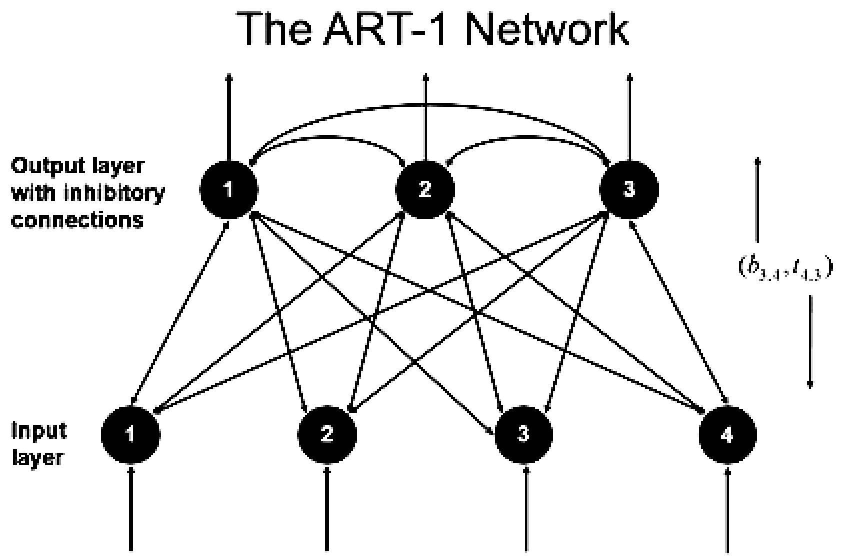


Figure 2.4 - *ART-1 network example*

Continuation of general research

To gain an understanding of these algorithms and how to use them, we looked at a current system called MATLAB. MATLAB has a toolbox similar to what is required for our task and so we used it a number of times to see what features we should implement. Also it shows how an industry standard piece of software includes all the different algorithms and other features that a neural network toolbox should include. This helped in building the core of our system but we didn't want to make a replica, we wanted create a toned down version which is simple,easy to use and has an improved GUI. We also looked at another system already in the market called Encog. While Encog appears a bit more raw and unpolished than MATLAB, it implements many of the features that MATLAB does. This suggests that these features that are consistent across both programs are mandatory in our own design. We incorporated these common features, but since these are made for very advanced users, requiring a steep learning curve for new users, we had to modify these existing designs to our own target audience. While fine tuning an AI algorithm - like choosing the activation function or the number of hidden layers can be useful, it's unnecessary for new users. As a result of this we decided to implement a pre-processor for the input data that tries to choose the best neural network setup for the given data, but still leaving the option for advanced users to fine tune the network. We also took a look at how Encog works internally as it is open source.

We concluded that it is very well designed and some of the ideas present there should be implemented in our software. This includes the processing of data which happens in several steps. This involves the randomisation, segregation, normalisation of data and a number of other steps. As a conclusion of our research on existing software we can say that these solutions require a deep understanding of neural networks and are difficult for new users. On the other hand they contain really good ideas and working patterns for advanced users which was a great source for that aspect of our implementation design.

After the researching about ANN and related algorithms used in our toolbox, the next chapter will present how the user interface was improved from the designing stage, and then some screenshots as well as updated software developing methodology.

Chapter 3: Updated Design of the System

Updated User Interface

With the aim of producing a clear, simple and clean project, a review and redesign of the architecture, framework and the GUI were needed. We invested a lot of time in brainstorming how to improve the system. Beside reliability and efficiency, choosing a familiar programming language for us was also important. Thus, the project was implemented primarily in Java. One of the major changes made to our plan was to make a standalone application (instead of a web interface). The main reason for the decision to develop a standalone system instead of a server was because we left it quite late to decide. Therefore we began to run out of time which left us going with the standalone application which was a much simpler concept to implement. In addition, we also solved the accessibility issue, as the user can run the application on any system that has Java installed.

Regardless of the changes, the main motivation was to create an attractive and easy-to-use system. The final decision was to use the JavaFx library to create the GUI of the system, because JavaFx can produce an attractive and lightweight program. The graphical user interface (GUI) is important to the usability of the software, the former GUI of the system was reviewed and improved. To improve the user experience of the system, the user has an option as to whether they would like to continue in advanced or basic mode which take different routes through the system. Advanced mode allows the user to configure the network how they wish while the basic mode simply gives them the option of choosing an algorithm for simplicity. Other than that, the user can now choose which columns of the data are input data, and which columns are output data for the neural network using the slider to specify. Last but not least, a help page is created to help the user obtain greater degree of help for using the system more easily.

Screenshots of the GUI design will be briefly presented below:

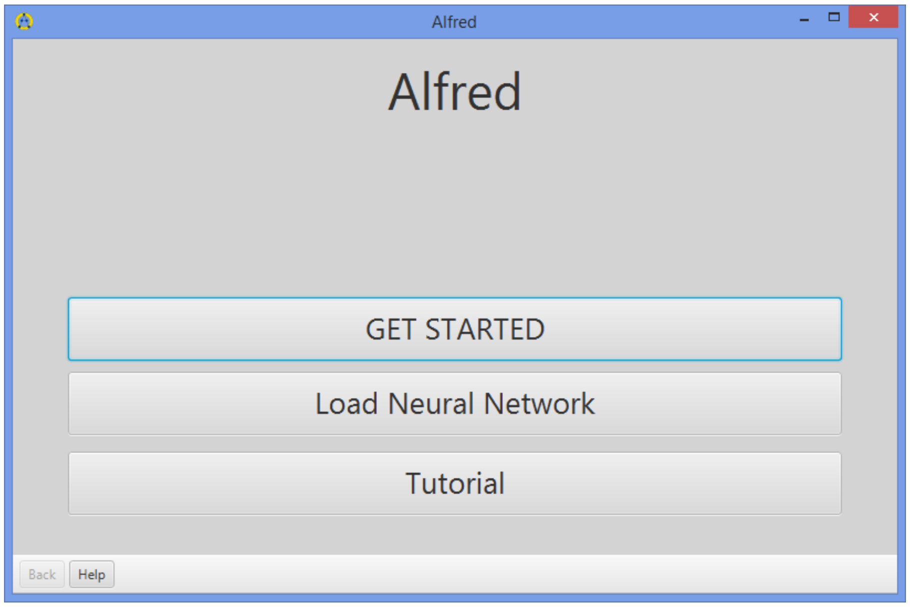


***Figure 3.1**: The first page of the toolbox where you can click "GET STARTED" to start using the program or you can load a previously saved neural network and there is also a tutorial mode in which the user is given help on each screen.*

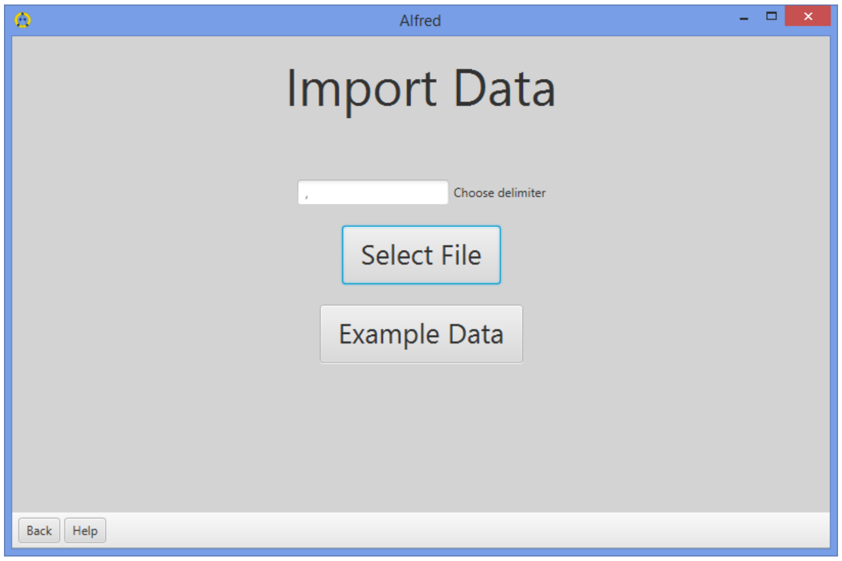
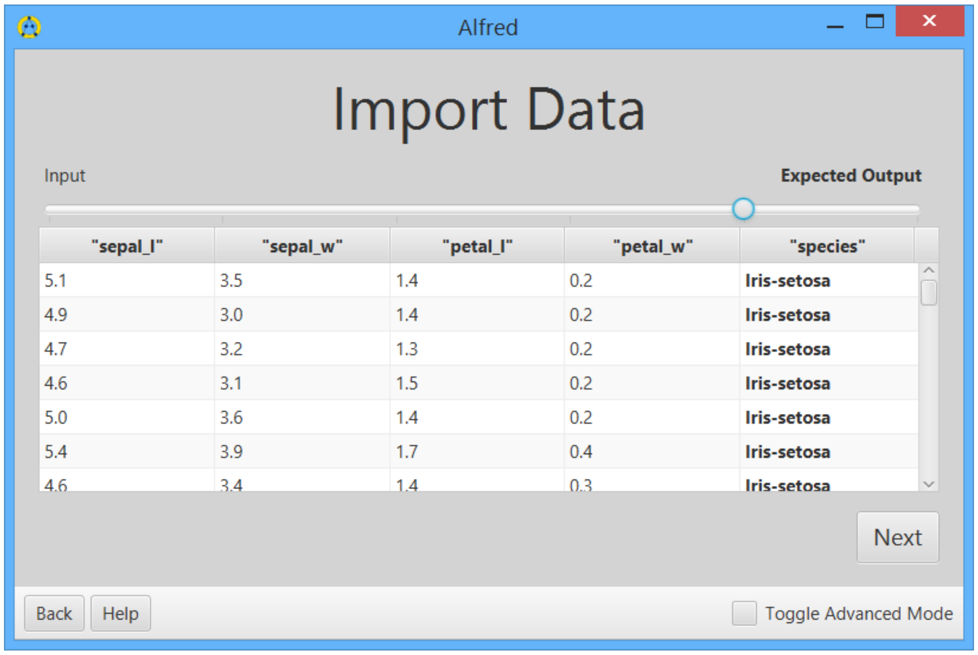


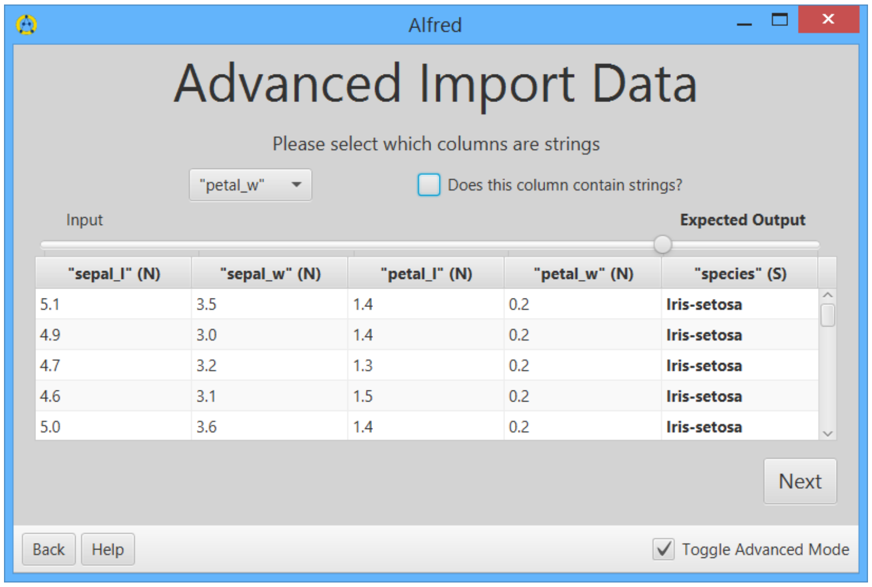
***Figure 3.2:** This is the import data screen from here you are able to select your file delimiter as well as load your selected. Once the file is loaded you taken to the next screen. The example data button can be clicked to load example data sets for testing.*


Alfred

Import Data

"sepal_l"	"sepal_w"	"petal_l"	"petal_w"	"species"
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3.0	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa
4.6	3.1	1.5	0.2	Iris-setosa
5.0	3.6	1.4	0.2	Iris-setosa
5.4	3.9	1.7	0.4	Iris-setosa
4.6	3.4	1.4	0.3	Iris-setosa

Figure 3.3: *This displays the user's chosen data. From looking at the UCI Machine Learning repository we were able to deduce that we had to allow the user a way to select the input columns of the data and the output. So we decided to implement a slider so the user can separate them efficiently. It was clear that although some of these example datasets follow a certain standard they still vary wildly in format.*


Alfred

Advanced Import Data

Please select which columns are strings

"petal_w"

Does this column contain strings?

"sepal_l" (N)	"sepal_w" (N)	"petal_l" (N)	"petal_w" (N)	"species" (S)
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3.0	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa
4.6	3.1	1.5	0.2	Iris-setosa
5.0	3.6	1.4	0.2	Iris-setosa

Figure 3.4:*This is the advanced mode of importing data compared with the previous normal importing data. Once users choose this mode by clicking the right bottom "Toggle Advanced Mode", advanced data selecting and importing will be shown.*

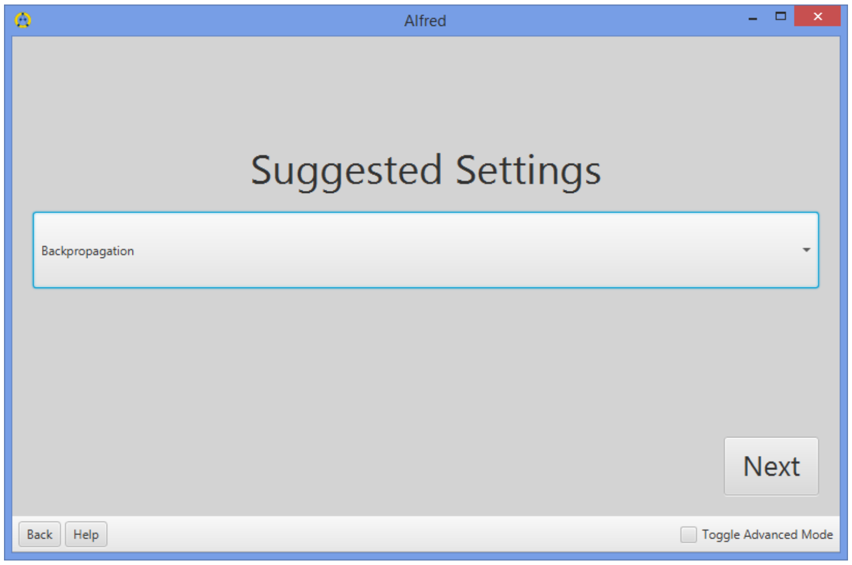
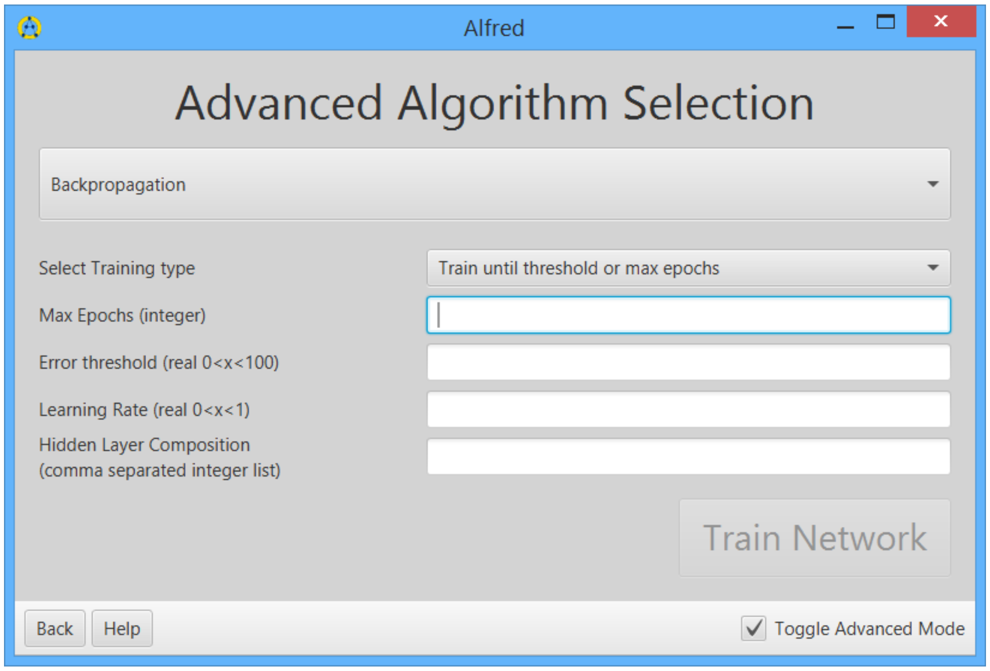


Figure 3.5: *This is the algorithm selection page: in this section users can choose different algorithms: if only training data is selected then unsupervised algorithms are suggested; if both training and target data is imported then supervised algorithms are suggested in the dropdown menu.*


Alfred

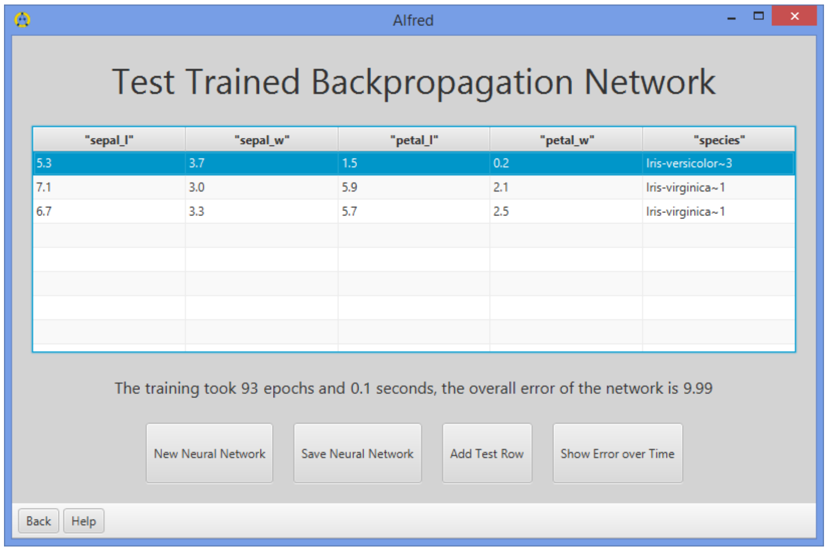
Advanced Algorithm Selection

Backpropagation

Select Training type	Train until threshold or max epochs
Max Epochs (integer)	
Error threshold (real 0<x<100)	
Learning Rate (real 0<x<1)	
Hidden Layer Composition (comma separated integer list)	

Figure 3.6: *This is the advanced algorithm selection compared with the previous basic one.*

Below are some results for the algorithms:

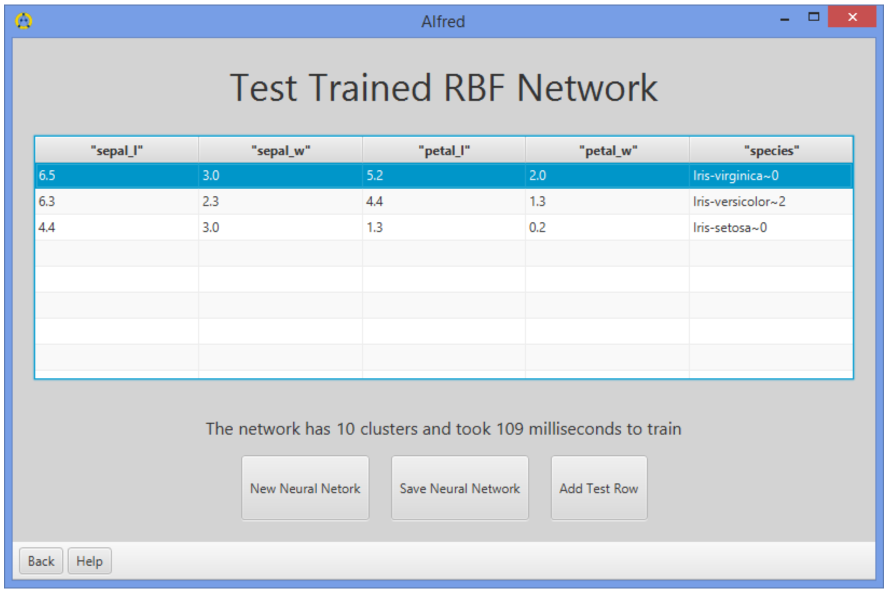

Alfred

Test Trained Backpropagation Network

"sepal_l"	"sepal_w"	"petal_l"	"petal_w"	"species"
5.3	3.7	1.5	0.2	Iris-versicolor~3
7.1	3.0	5.9	2.1	Iris-virginica~1
6.7	3.3	5.7	2.5	Iris-virginica~1

The training took 93 epochs and 0.1 seconds, the overall error of the network is 9.99

Figure 3.7: *The result page shown after training a Back-Propagation network. This image specifically shows a BP network trained with the Iris dataset which trained for 93 epochs, took 0.1 seconds to train and has a error of 9.99%. You can add more test rows, save the network and show a error over time graph for the network. Changing values in the table will recalculate the output.*

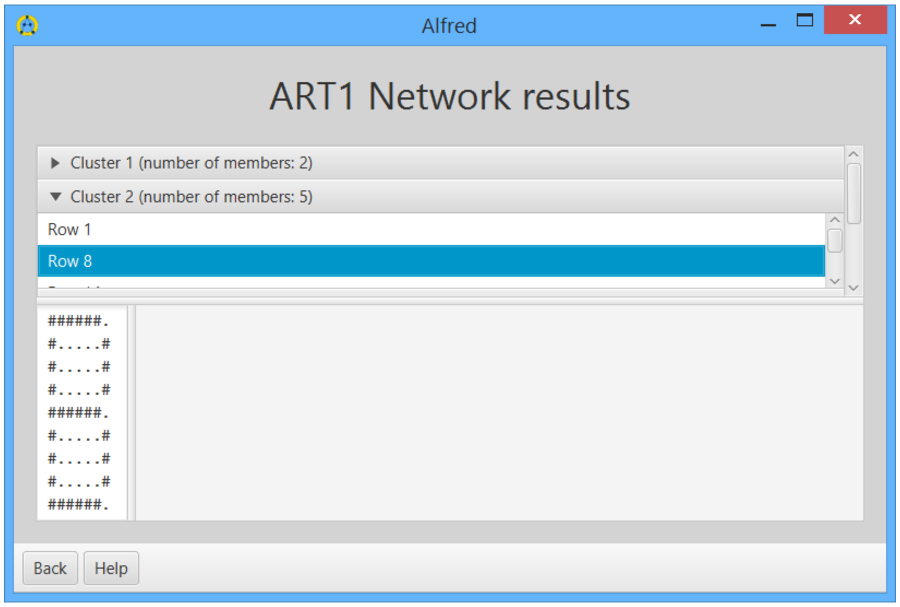


Test Trained RBF Network

"sepal_l"	"sepal_w"	"petal_l"	"petal_w"	"species"
6.5	3.0	5.2	2.0	Iris-virginica~0
6.3	2.3	4.4	1.3	Iris-versicolor~2
4.4	3.0	1.3	0.2	Iris-setosa~0

The network has 10 clusters and took 109 milliseconds to train

Figure 3.8: *The result page shown after training a Radial Basis Function network. This image specifically shows a RBF network trained with the Iris dataset which had 10 clusters and took 109 milliseconds to train. You can add more test rows and save the network. Changing values in the table will recalculate the output.*



ART1 Network results

Cluster 1 (number of members: 2)
Cluster 2 (number of members: 5)
Row 1
Row 8

```
######.
#.....#
#.....#
#.....#
######.
#.....#
#.....#
#.....#
######.
```

Figure 3.9: *The result page for ART1 displays all the clusters created in an accordion, clicking a cluster reveals which rows were put in that cluster. Clicking on a row will display the binary data in the box below "0" is represented with a . and "1" with a #. The data shown is of binary image data of character sets and a "B" is shown in the image.*

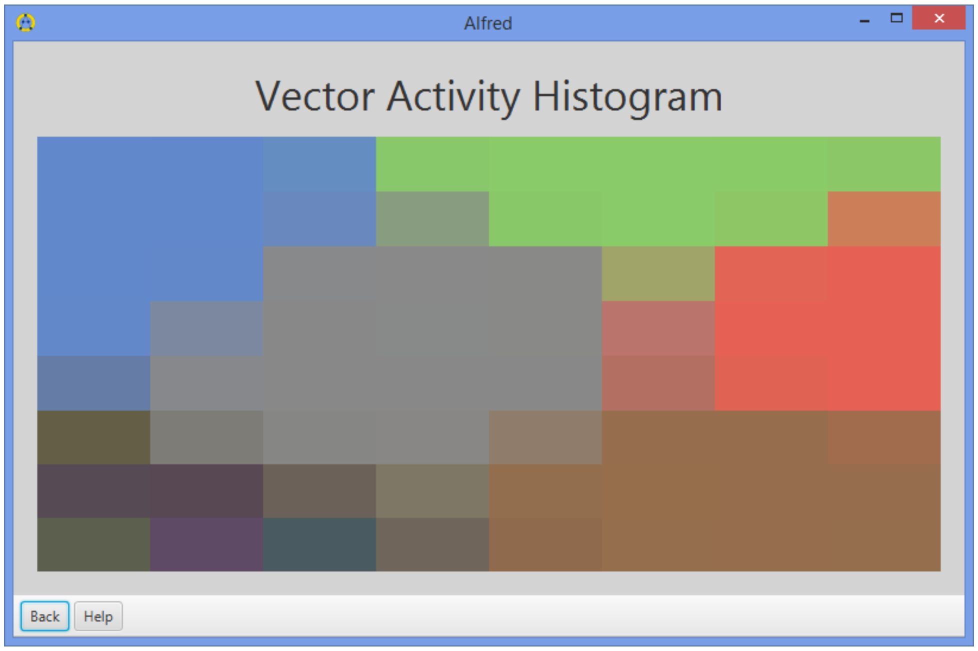


Figure 3.10: *This is the result of a SOM output in our system. For this specific output a five random RGB values were used as the input with a lattice dimension of 8x8, epochs of 1000 and learning rate of 0.7.*

System Requirements

Functional Requirements

The updated functional requirements can be seen in Appendix E.

Non-Functional requirements

The updated non-functional requirements can be seen in Appendix F.

Updated methodology

The software engineering methodology we chose for the first semester was Extreme Programming (XP) due to the fact that at that time the specific requirements needed to be updated. The reason choosing XP was that it allowed changing requirements. However, when actual programming started, the methodology used was more like a combination between spiral model and XP. Since the user requirements in Appendix E and F were specified and we had a strict Gantt chart with deadlines and development in stages constantly looking over previous code meant that the spiral model was the right choice. As this approach helped us get the work done in the end.

After knowing the updated design of our system, the next chapter will present how the actual programming was implemented and show detailed implementation.

Chapter 4: Implementation of the System

Class Structure

There are a number of learning algorithms that have been developed, however we picked four of the most common and relevant algorithms for this project. The Four algorithms implemented in this project are Backpropagation (BP), Self Organizing Map (SOM), Radial basis function (RBF) and finally Adaptive Resonance Theory (ART1).
The implementation of the algorithms of the system will be showed as following:

1. Backpropagation (BP):

Backpropagation requires training and target data. Each column of data is normalised so that all the data in that column is between 0-1. There are several layers with every neuron in the layer connecting to all the neurons of previous layers. Every connection has a random weight and bias assigned to each neuron when the network is created. Working with a neural network using BP includes two steps :

- Training - gathering inputs and their corresponding outputs.
- Running - by providing the input to obtain the output.

An input is given to the network and then its obtained output which might not alway be accurate. The accuracy of the output during the running segment depends a lot on the configuration and data used in the training segment. The training segment consists of adjusting the weights and bias values of each neuron in the NN except for neurons in the input layer.

Four sub-steps are required to make BP work and these are:

1. Feeding The Inputs
2. Finding the output of the network
3. Calculating The Error or Delta
4. Adjusting The Weights and Bias

Step 3 should begin by calculating the error or delta value for each neuron. Then this delta value will be used to calculate the error or delta of the neurons in the last hidden layer. Then the delta value of all neurons in the last hidden layer will be used to help calculate the error or delta in the second last hidden layer and so on. This is process is carried out until the first hidden layer is reached. No calculations will be made for the input layer. A neuron in any hidden layer uses the delta of all the connected neurons in the following layer and also the conforming connection weight to find error factor. A lot ideas for the implementation of Back-Propagation came from (Madhusudanan 2009).

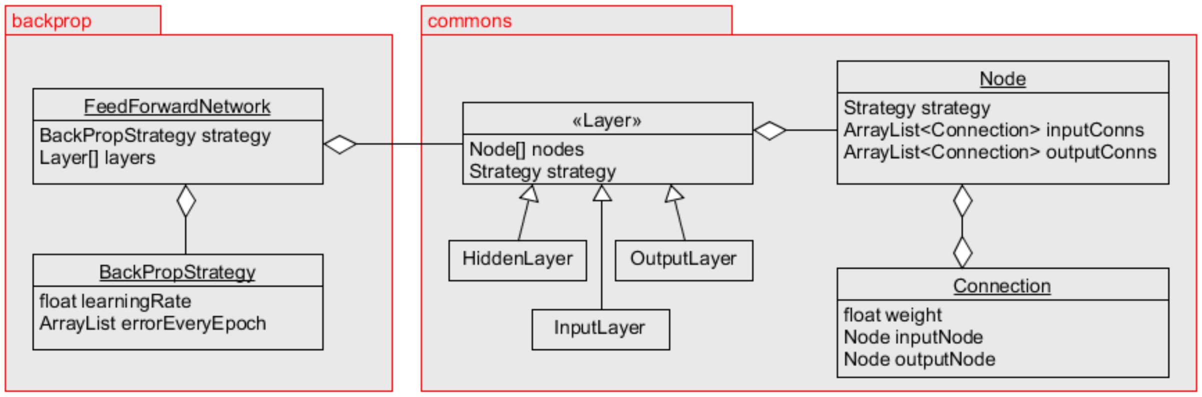


Figure 4.1 *- The class diagram showing how the Back-Propagation network is structured in our implementation.*

2. Self Organizing Map (SOM):

Due to the unsupervised learning of the SOM algorithm it is quite different to the supervised style of other algorithms, the classes do not inherit from the back propagation classes. Instead, it is more of a standalone.

Firstly the KNode class was created, which basically acts as a vector of float arrays. When a node is created, a size is passed into the constructor which lets it know how many weights the node needs to have. It then randomizes that number of values and stores them in the vector. The class then has a method to access each weight and set them if need be.

A KLattice class was then created as it was the next essential part of the implementation. This class takes a height and width integer in the constructor and creates a 2D array using these values as dimensions. At each point of the array a KNode is created and stored. This creates the effect of a lattice with nodes at each point containing a number of weights. Most of the functionality of the algorithm is also present in this class as it has easier access to the nodes this way. This includes functions for finding the BMU, calculating the neighbourhood radius and finding all values in that radius, and calculating the Euclidean distance (Described in research section of SOM).

Next was the input layer class which is implemented as a vector of vectors that contain weights. In context to the SOM algorithm it is a vector that contains KNodes. Then finally the two classes that connect all of the others together are the main and SOM class. The main creates the SOM network and the SOM class takes the passed in input and calls each function in the correct order for a specified number of iterations.

Originally we planned to produce a scatter graph for this algorithm, however we have now decided to produce a vector activity histogram. This selects an input vector and compares its distance from each node in the lattice and based on its similarity the colour is altered. Areas of similar colours represent similar weight vectors.

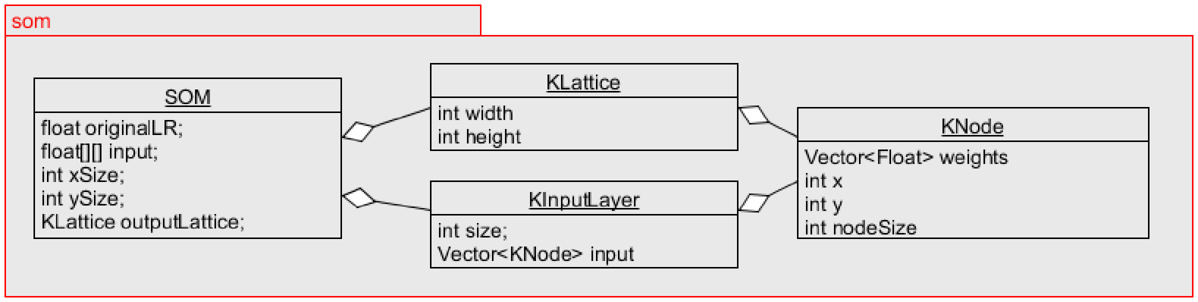


Figure 4.2 *- The class diagram showing how the Self-Organising Map network is structured in our implementation.*

3. Radial basis function (RBF):

Radial basis function networks (RBF) effectively perform pattern classification by transforming the problem into a high dimensional space in a nonlinear manner. The result is a linearly separable problem that is easy to solve. RBF networks are used for supervised learning so in many ways it is very similar to backpropagation (BP) in that the network is constructed using multilayer perceptrons (MLP) this means that some classes are able to be reused in the RBF network implementation. One way in which they differ is the hidden nodes instead of being able to have multiple hidden layers RBF uses only one hidden layer. The hidden nodes use Gaussian functions as the activation instead of using activation functions like sigmoidal. A typical RBF network structure can be seen in figure 4.3.

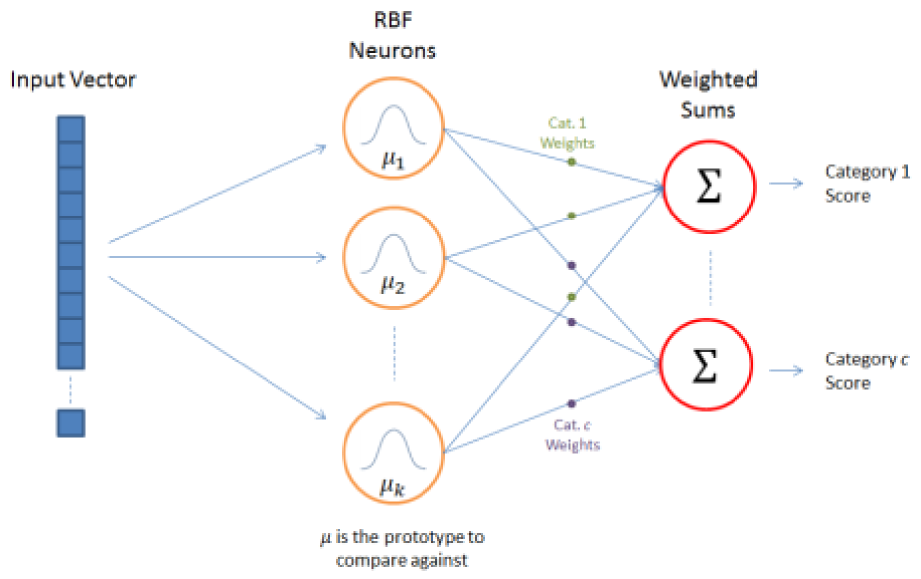


***Figure 4.3** - Structure of a RBF network*

The classes created for the RBF network can be seen in Figure 4.4. The way to structure the network was to reuse our InputLayer class from BP. Since the hidden layer works in totally different fashion to BP, creating an RbfHiddenLayer class which extends Layer and consists of RbfHiddenNode's which call their functions from RbfStrategy were decided. The final layer is also a custom RbfOutputLayer consisting of regular nodes from BP. This due to the fact the only difference between BP and RBF output layers is the summing functions which in RBF networks act as the activation function this function is called from RbfStrategy. The inputs are fed into the network a clustering algorithm is used to calculate cluster centres and these are the set for each hidden node. The outputs from the hidden node activation functions are then calculated for the entire training data and all outputs are stored in a matrix. A pseudo inverse is then performed on the matrix this is multiplied with the target outputs and the resulting matrix is then transposed to give the weights of connections between the hidden and output layer. Once this is finished the network has been trained and can be used to run new inputs and see their output.

So the parameters going into the hidden node activation functions are the inputs themselves and the $\boldsymbol{\mu}_1 \ldots \boldsymbol{\mu}_k$ from Fig 4. μ is the centroid of a cluster and each hidden node (k) represents a cluster, there are many ways to determine these cluster centroids one is to choose them randomly. But a more accurate way is to use a clustering algorithm such as K means clustering. K means clustering works by initially choosing random centroids for k number of clusters adding the x_{inputs} (points) to the clusters based on which centroid they are closest to. The centroids are also periodically updated based on the points in the cluster. For our implementation, we opted for variation on k means clustering called Greedy Variance Minimization (GVM) fast spatial clustering which was developed by [Gibara 2011]. One of the reasons to choose this algorithm is that Tom Gibara provides a very simple and easy to use java library that works natively with floats in java. This is in contrast to k means clustering which was unable to find a simple library for. Instead, there are only large complex machine learning api's which provide k means clustering to native file formats, one such example is the Weka machine learning java api which requires data in its native ".arff" file format. An advantage of GVM as well is that its memory usage is constant and independent of the number of points clustered. A full comparison of GVM to k means is given by Gibara on his website.

After the GVM algorithm has produced the centroids for each hidden unit the activation functions are calculated, the Gaussian function we used for the hidden nodes was:

$$\Phi(x) = e^{-\beta||x-\mu||}$$

β is the cluster variance of each cluster which is essentially how spread out the points are in a cluster. In the Gaussian equation it limits the width of the bell curve generated by the function. $||x - \mu||$ is the Euclidean distance between the input vectors and the centre vectors.

Once the training data has all been run through the network all its outputs need to be stored to perform matrix transformation with so that the connection weights between the hidden and output layer can be calculated. The summing function of the output nodes looks like:

$$F(x_i) = \sum_{i=1}^{N} w_i \Phi(x_i)$$

$$F(x_i) = d_i$$

N represents the number of hidden nodes connected to this output node, w is the connection weight between the hidden and output node and d_i is the target output. We could write this relationship as: $\Phi w = d$

From this we can calculate the outputs using fast linear matrix inversion techniques so that:

$$w = (\Phi^{+} d)^{T}$$

Φ^{+} represents the pseudo inverse of the activation output matrix, d being target outputs and the T represents that the resulting matrix in the brackets needs to be transposed. Most of these equations were taken from the (Bullinaria 2004) lectures on RBF networks.

For our implementation, a library called JAMA (JAMA 2012) was chosen to use. JAMA was designed to be used as standard matrix library for Java, so it was designed to be understandable by non-experts, which was important for us. It also provides all the standard operations will be required for the weight calculation. Also an extra column was added to the Φ (activation function output) matrix all with the value of 1, this column is used to calculate bias weights for each output node. This compensates for the difference between the average value across the data set of the basis function activations and the corresponding value of the target.

Once the weights have been calculated the network is finished training and new data can be run through the network. To do this first need to get the RbfNetwork object call the feedInput method then the feedforward method and finally get its output by calling getOutput.

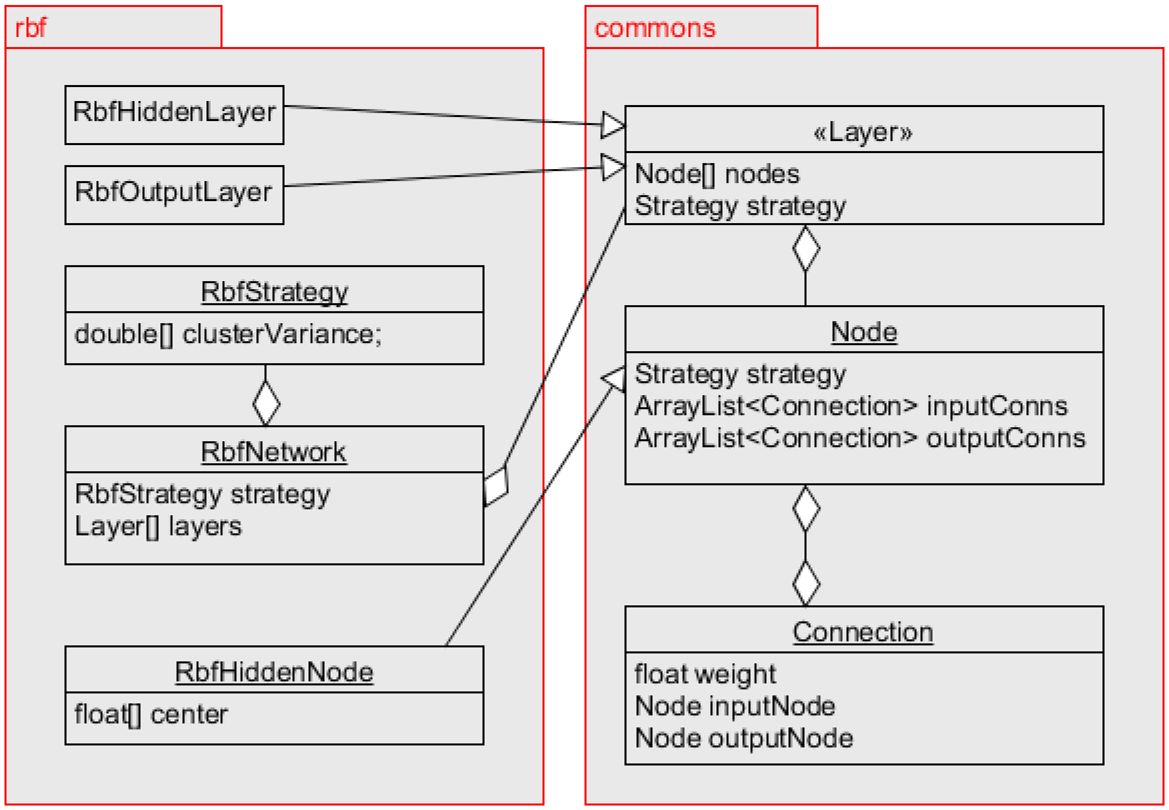


Figure 4.4 *- The class diagram showing how the Radial Basis Function network is structured in our implementation.*

4. ART

The Adaptive Resonance Theory (ART) network works as follows. It has two layers: an input layer and a clustering layer (commonly referred to as F1 and F2 layers respectively). These layers are fully connected both ways, that is, each node on the two layers has a bottom-up and top-down weight connecting it to each other node on the other layer. The top-down weights act as a long-term memory and bottom-up weights act as short-term memory. The network is trained as follows: For each set of input data the input layer is presented with it, then using this the output of the clustering layer's output is calculated. On the output layer the node with the highest output is selected. If this node's similarity is higher than the vigilance factor then the weights are updated connecting to this node (both bottom-up and top-down), making it more similar to the dataset. This also means that this dataset belongs to the cluster represented by the winner node. Otherwise the node gets rejected and the process starts again from finding the maximal node. If all nodes are rejected then a new node must be created. This newly created node is going to learn the input pattern that rejected all the previous nodes.

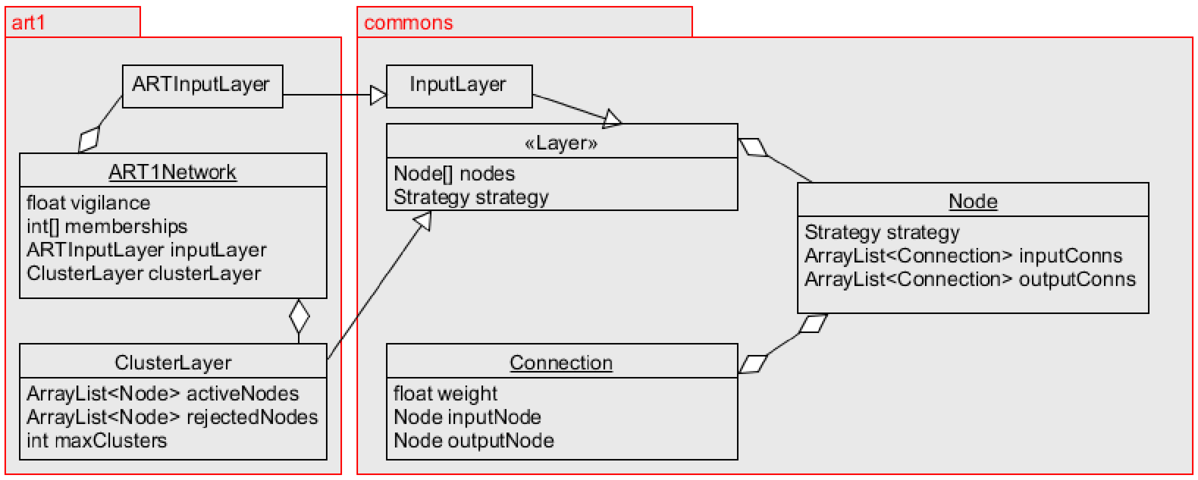


Figure 4.5 - *The class diagram showing how the Adaptive Resonance Theory (1) network is structured in our implementation.*

5. GUI

Since we were using Java for our ANN implementations we had opted to use JavaFX for our GUI implementation. The GUI part of system was what would tie all of our ANNs together, now JavaFX does use the Model-View-Controller paradigm, and this can be seen in Fig 4.6.

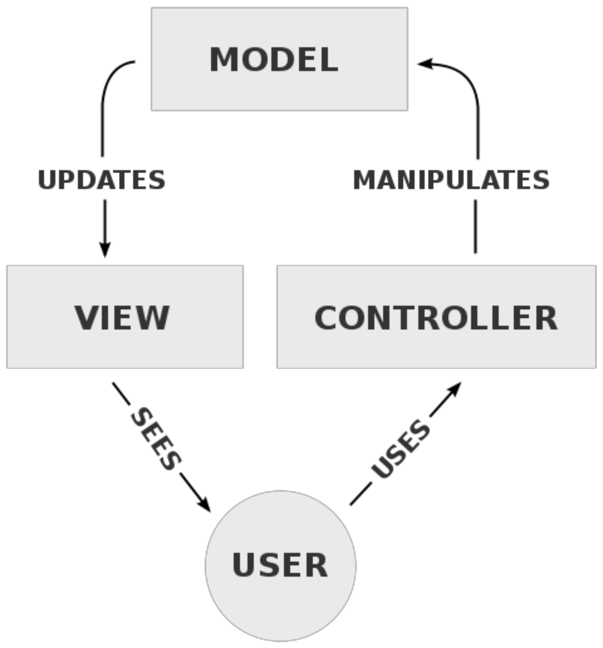


Fig 4.6 *- This diagram illustrates the Model-View-Controller paradigm and shows the parts of system the user interacts with and the parts displayed to them*

The way in which JavaFX handles this is to have Views which are FXML documents that layout the structure of the GUI. Then each view has a controller assigned to it. This controller has methods which are called when the page is loaded and it calls methods when actions are performed on the view. The models exist so that JavaFX can persist objects across multiple views and the models in our project were how we persisted the neural network states across the views. Models use static variables as objects cannot be passed between controllers, this means every time that the model is created the states of it variables persist. So for our project we have a class called Model which holds a bunch of different variables, such as booleans, to check if advanced mode or the tutorial are enabled. It also holds the normalised training and target data as well as the raw data that has been read in. Additionally Model holds our Normaliser and FileParser objects which have states that need to persist for example the normaliser holds a hashMap of strings so that when denormalizing, the string can still be found from the float value generated. Other models that we use are CreateAndTrainBP, CreateAndTrainRBF and CreateAndTrainART these are used to persist each neural network across the controllers. They save the ANN's as statics as well as store a few different variables such as duration to train and number of epochs. For SOM it turned out to be easier just to make the variables it uses static so effectively the SOM network acts as it's own model.

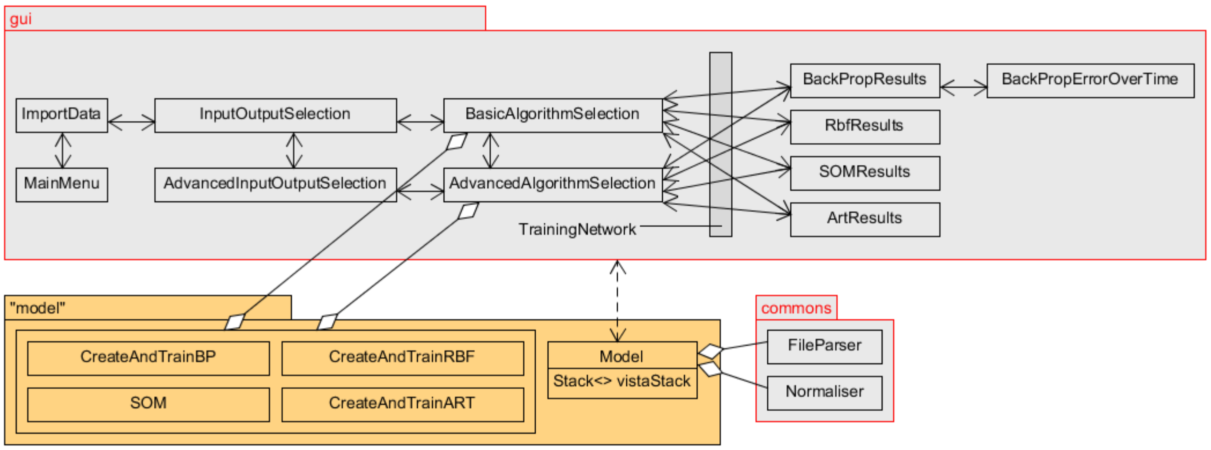


Fig 4.7 *- This is the simplified class diagram for the gui package. Each class in the gui box represents an interconnected view and controller. The TraingNetwork class is the loading screen, so it is represented differently as it is just an intermediate step. The "model" box is marked with a different colour as it is not an actual package but these classes act as models.*

Chapter 5:Testing of the System

Due to the random way the algorithms work each time specific tests couldn't be written for an entire NN algorithm. Hence junit tests were created for classes of each algorithm. These can be seen in Appendix D for testing. These test the internal functions/methods that compose the algorithms. As seen in Appendix D all tests passed.

Back-Propagation (BP) and Radial Basis Function (RBF) - benchmark testing

Due to the similarity of BP and RBF in regards to the problems they work on we decided to use the same testing methodology for both. Now although there are a few ways of effectively benchmarking neural networks we quickly discovered this would be almost impossible for our project. This is because to benchmark effectively you need to have complete control over every parameter of the neural network and just due to the amount of control we give a user this would be impossible in our toolbox. So instead of extensive benchmarking we instead opted to run RBF and BP networks on both our system and MATLAB's with as close to the same settings as we can manage and would compare their results in relation to the expected results. The dataset we decided to use for this benchmarking was the Iris data set from (Lichman 2013), which is a widely used dataset in machine learning.

The first network we tested was BP for this network we choose to run the network for 10000 epochs regardless of its error. For the layer composition we opted for 2 hidden layers with 3 nodes each and a learning rate of 0.01.

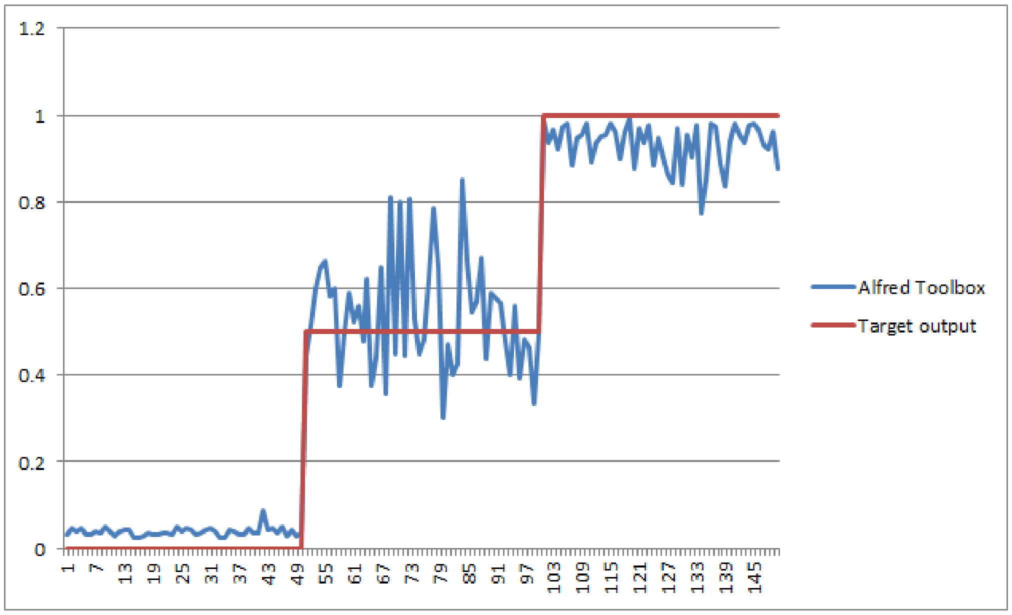


Figure 5.1 *- Back-Propagation results from running the Iris data set through our Alfred toolbox. Red line represents the target output. The blue line represents the actual output of the Alfred toolbox. The x axis represents each set of inputs.*

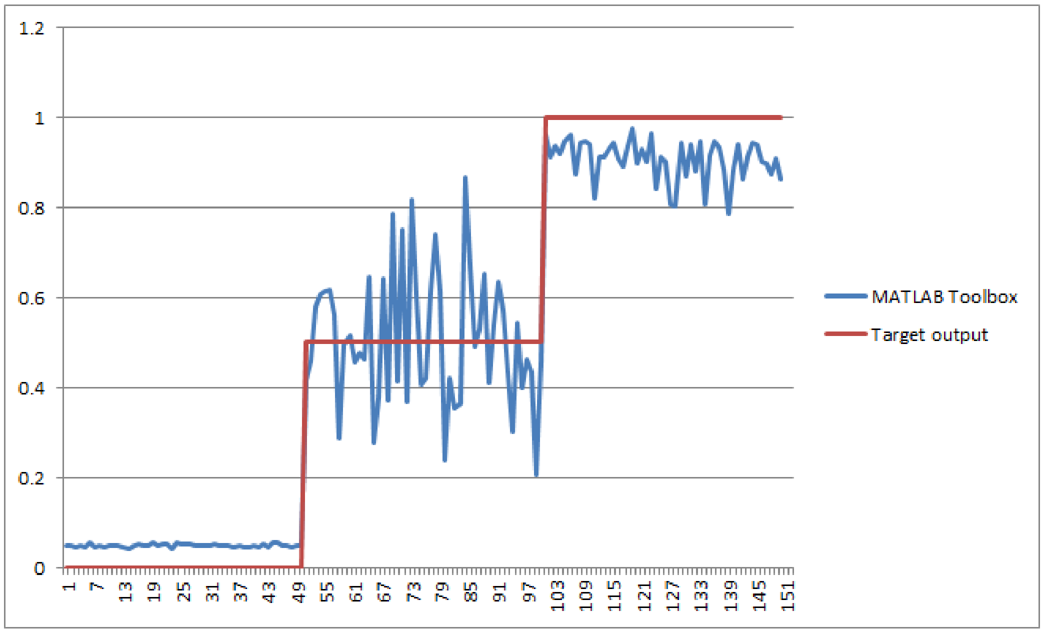


Figure 5.2 *- Back-Propagation results from running the Iris data set through the MATLAB toolbox. Red line represents the target output. The blue line represents the actual output of the MATLAB toolbox. The x axis represents each set of inputs.*

As you can see from Fig 5.1 and Fig 5.2 our Alfred toolbox performs almost identical to the MATLAB toolbox on a target output of 0.5. For a target output of 0 Alfred actually appears to be closer to the target output than MATLAB and for a target output of 1 Alfred appears to consistently be closer to the target output. In terms of classification both appear to perform almost identically.

Next was the RBF benchmarking, for our implementation the only selectable parameter is the number of clusters (number of nodes in the hidden layer) so we tried to match MATLAB's implementation as closely as we could to ours. The number of clusters chosen was 9 was this produces a fairly accurate result and the Iris dataset was used again.

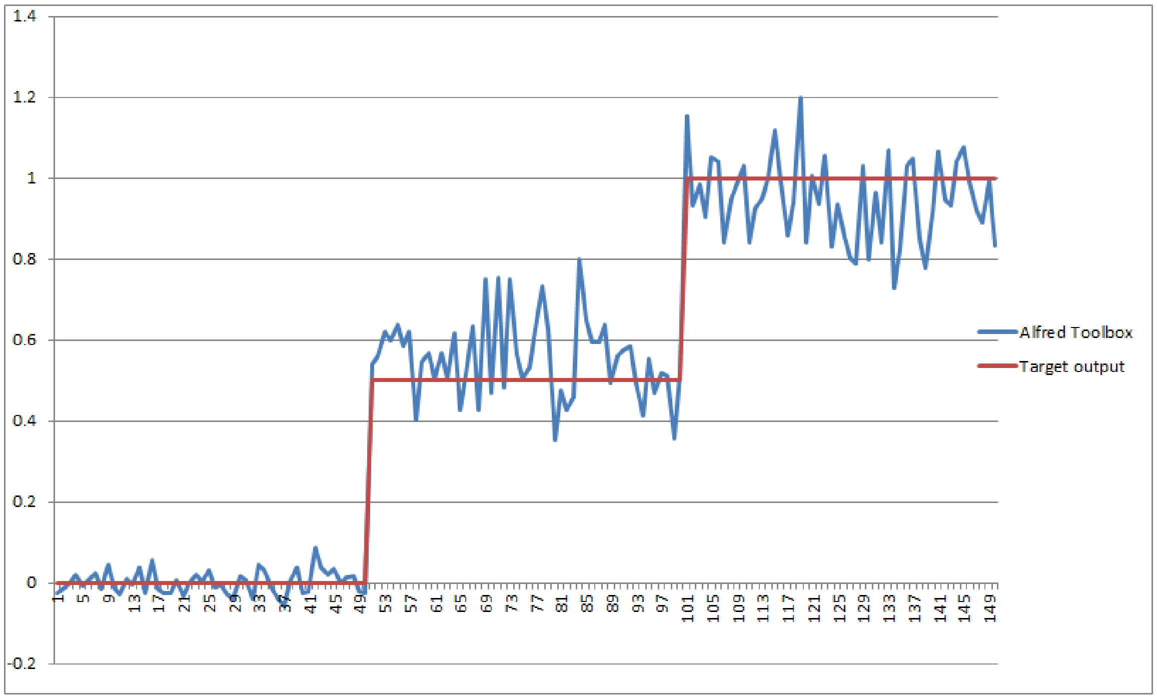


Figure 5.3 *- Radial Basis Function results from running the Iris data set through the Alfred toolbox. Red line represents the target output. The blue line represents the actual output of the Alfred toolbox. The x axis represents each set of inputs.*

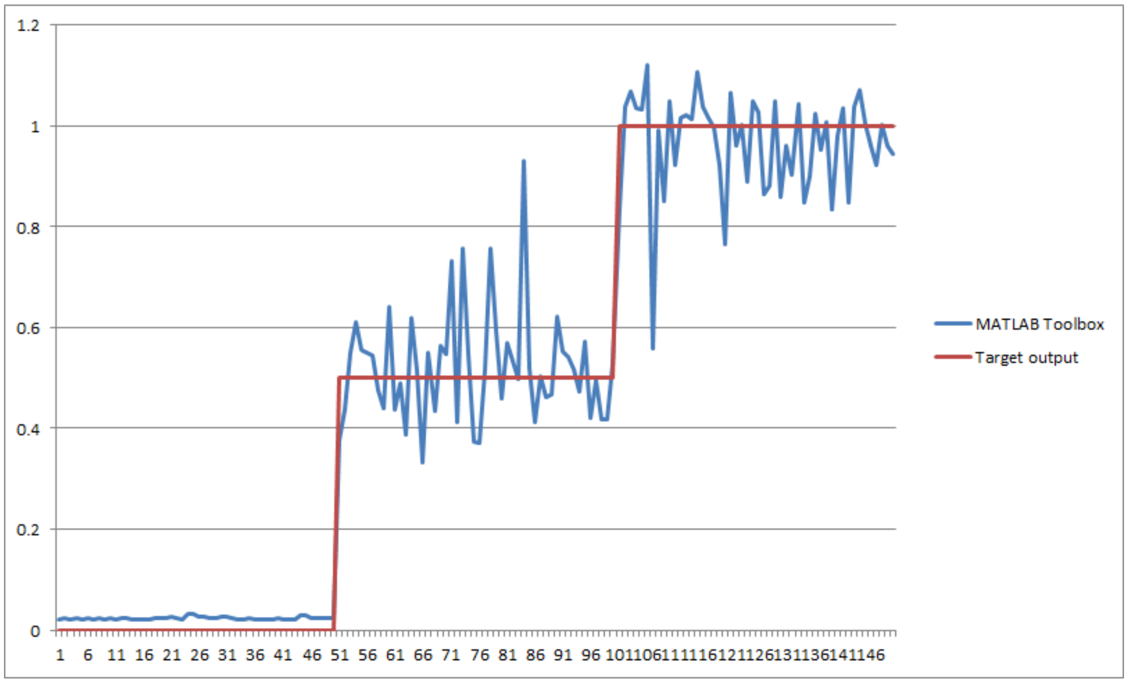


Figure 5.4 *- Radial Basis Function results from running the Iris data set through the MATLAB toolbox. Red line represents the target output. The blue line represents the actual output of the MATLAB toolbox. The x axis represents each set of inputs.*

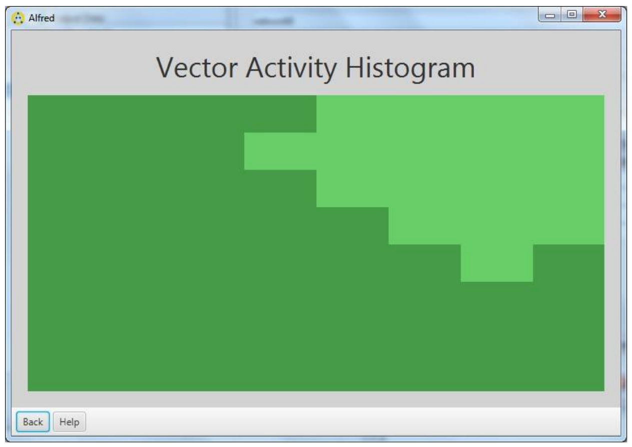
From Fig 5.3 and Fig 5.4 we can see that for a target output of 0 MATLAB's implementation appears to be more stable with less variance. However when we look at a target output of 0.5 and 1 we actually see that the Alfred implementation has less variance than MATLAB which tends to indicate better classification. However we can see that both have fairly similar results and both appear to be accurate to a certain degree but not nearly as close as are results from BP.

From these results we feel that both implementations of RBF and BP do their intended job to a acceptable level of accuracy. Although these tests are not hugely scientific we feel we can deduce their overall performance from them which we are happy with.

Self-Organising Maps (SOM) - benchmark testing

While we aimed to rigorously benchmark every algorithm to ensure a high level of quality in the implementation, SOM is quite a difficult algorithm to test. This falls down to the fact that the output visualization of it is very arbitrary and it involves randomised values which lead to somewhat different results every time it is run despite using the same data.

That being said, we tried comparing the output of our algorithm to MATLAB's SOM algorithm and these were the results of the IRIS data set from the UCI repository (Lichman 2013) and the configuration for both algorithms was 1000 epochs with a learning rate of 1:



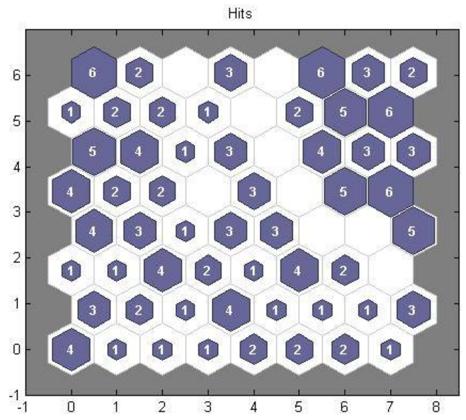


Figure 5.5 and Figure 5.6 respectively *- Alfred's SOM output with IRIS in a Vector Activity Histogram and MATLAB's SOM output with IRIS in a Hit Histogram.*

It is clear to see that while they are not exactly the same (Figure 5.5 and Figure 5.6) there is a similar trend to the pattern conveyed in the images. There is a cluster in the top right of each image and a larger cluster surrounding it. The subtle differences mostly come down to the fact that the lattice is originally randomized for its weights, meaning every time the algorithm is run it can be different. This suggests that training occurs in a similar way to that of MATLAB and is working effectively.

When our SOM algorithm is run with a large randomly generated RGB colour data set and set to have much larger dimensions, it produced this (See figure 5.7):

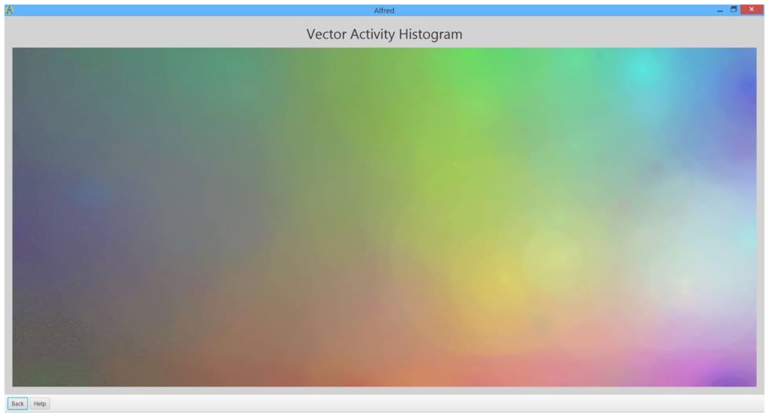


Figure 5.*7 - Alfred’s SOM with random RGB values*

This demonstrates the sorting capability of the algorithm as it is able to sort a large number of colours into relatively specific groups.

Knowing basic ideas about testing the system, the final chapter will give some comments to reflect the whole year’s developing process.

Chapter 6: Reflective Comments on the Project

In this chapter, we will reflect on our experience of participating in G52GRP. In general we consider the project to be a success as we have developed a final product that meets the requirements in Appendix E and F, which were initially specified. We have written our views on major areas of the project in which we had issues as well as our achievements.

Achievements according to Requirements

With reference to the requirements in Appendix E and F, achievements are listed as below:

- Developed an easy to use application which is compatible with any operating system.
- Developed an interface that hides away the complex nature of neural networks from a non experienced user while also providing advanced options for advanced users.
- The interface allows the user to train and test a neural network.
- The interface allows the user to import any sort of training data or gives the option to choose example data.
- The interface displays graphs for statistics regarding the training of the network
- Developed a tutorial which is easy to follow and relevant tutorials are provided for each section of the interface.
- Developed an interface that doesn't require an extensive amount of time.

Problem Encountered

GUI-JavaFX

Due to the nature of javafx, parsing a variable through the application into a controller was a problem. As we had written code for the algorithms in an object oriented manner having the ability to parse variables was vital. This made us consider using swing instead of javafx but we were able to figure out a compromise using getters and setters to parse variables to solve the problem. Although the solution was tedious, the compromise was vital as some group members were already familiar with javafx. Hence, this meant we didn't need to learn swing or any new language therefore saving time.

SOM Issues

While constructing the Self Organizing Map algorithm we faced a few problems that were eventually overcome. The main problem that prevented the algorithm from working correctly was that the best matching node function did not actually find the best node, it would just select the first node of the lattice. This was problematic as the clustering of colours is dependent on altering a radius around this node, so it must be accurate. The problem was that the best distance variable was initialized at a very low value so every distance it was compared to was too high. To fix this we initialized the value as MAX_VALUE, which prevented this problem from happening again.

Code Sharing

Initially all coding was being done in one directory and this caused problems because any accidental changes could affect the working copy. As progress was made and work was allocated separately this problem was brought to the groups attention. . Hence two sub-directories were introduced, branches and root(working copy). After a series of discussions, the team realised that not using the SVN was ineffective and we made sure every member of the group knew how to operate it. It proved to be crucial to the success of the project as the team was much more effective in the development of the code and there were far less issues regarding code conflicts.

Reflection on the Project

Looking at the different members of our group and analysing the compatibility of our different skills and backgrounds, it is obvious there were some barriers between the members of the group, especially at the beginning. For example there was a significant cultural and language barrier between the local and international members at first, however over time the international group members were able to work well within the group. At the start of the project we decided to meet once per week and spend time doing research individually. While members of the group finished their tasks diligently, the group lacked a consistent idea of how the final product should look. At some stages of the project the group was demotivated by the difficult project in hand, yet every member continued on and was committed to deliver the end product by the deadline.

The communication between the members of the group was good despite the language barrier. A facebook group and scrum board was created to distribute tasks, discuss new ideas, report the progress across the group and generally keep in contact. We also developed a GANTT and PERT chart to keep track of our progress and maintain a common goal within the group. Other than that, the communication with the supervisor was very useful as he would always make time for the group and give constructive advice on both technical elements and the management of the team. In conclusion, the group dynamic improved over the duration of the project. In the first semester, we were lacking commitment to the project due to a number of deadlines for other modules. This left us with less time to work on the project over the whole year but in the second semester all group members invested a lot of time into the project and made it a priority over other deadlines.

To conclude, reflection over the whole project showed certain areas of success. Our group was able to create a detailed specification about the project including basic UI design and clear requirements in the first semester successfully. As for the second semester, although we faced stress since the deadlines were really close, efficient programming and strict adherence to the schedule helped stimulate us into finishing the task on time. Luckily the programming process was efficient; usually improvements and progress was made after each weekly formal meeting. So, in the later term we finished both the software and the report, and especially the software met the requirements well. Our project management of the beginning of the implementation stages was to blame, but the effective recovery allowed us to increase the work rate and get back on schedule. The implementation of the software was fairly successful in the fact we implemented our functional and non-functional requirements to a good standard. By the end of the project we had come together as a team with good communication, specific roles and a friendly atmosphere. Each group member finished his or her own tasks and completed the project to the best of their ability, each contributing in some way to the final product.

Appendix A--References

Demuth, Howard., Beale, Mark. and Hagan, Martin.(2008) "Neural network toolbox™ 6." *User's guide*. [Online] Available at: "http://kashanu.ac.ir/Files/Content/neural_network_toolbox_6.pdf"

Maltarollo, Vinícius Gonçalves., Albérico Borges Ferreira da Silva, and Káthia Maria Honório. (2013) *Applications of artificial neural networks in chemical problems*. INTECH Open Access Publisher.

Fauske, Kjell Magne. (2006) "*Example: Neural network*" [Online] Available at: "http://www.texample.net/tikz/examples/neural-network/ "

McCulloch, Warren S., and Pitts, Walter . (1943) "A logical calculus of the ideas immanent in nervous activity." *The bulletin of mathematical biophysics* 5.4 : 115-133.

Hopfield, John J. (1982) "Neural networks and physical systems with emergent collective computational abilities." *Proceedings of the national academy of sciences* 79.8 : 2554-2558.

Werbos, Paul J. (1990) "Backpropagation through time: what it does and how to do it." *Proceedings of the IEEE* 78.10 : 1550-1560.

Heaton, Jeff (2007) "Implementing The Kohonen Neural Network": "http://www.heatonresearch.com/articles/6/page3.html"

Buckland, Matt (2004) " Kohonen's Self Organising Feature Maps": "http://www.ai-junkie.com/ann/som/som1.html"

Ringwood John, Galvin Gareth (2002). "*Artificial Neural Networks*" Dublin, Ireland: Dublin City University, School of Electrical Engineering. [online] Available at: "http://annet.eeng.nuim.ie/intro/course/chpt4/Chapter4.shtml"

Bullinaria, John A. (2004). "*Introduction to Neural Networks*" Birmingham, UK: University of

Birmingham, School of Computer Science. [Online] Available at: "http://www.cs.bham.ac.uk/~jxb/NN/"

Madhusudanan, Anoop (2009): *"Designing and Implementing A Neural Network Library For Handwriting Detection, Image Analysis etc."* [Online] Available at: "http://www.codeproject.com/Articles/14342/Designing-And-Implementing-A-Neural-Network-Librar"

Gibara, Tom. (2011) "*GVM: Fast Spatial Clustering*" [Online] Available at: "http://www.tomgibara.com/clustering/fast-spatial/"

JAMA (2012) "*JAMA* : A Java Matrix Package" [Online] Available at: "http://math.nist.gov/javanumerics/jama/"

Grossberg, S. (1976). Adaptive pattern classification and universal recoding, I: Parallel development and coding of neural feature detectors. *Biological Cybernetics, 23, 121-134.*

Lichman, M. (2013). UCI Machine Learning Repository. Irvine, CA: University of California, School of Information and Computer Science. [online] Available at: "http://archive.ics.uci.edu/ml"

Appendix B--Images

Figure1.1"http://electronicsnewsline.com/819/biological-neural-network-and-artificial-neural-network-a-comparison.html"
Figure 1.2 "http://www.texample.net/tikz/examples/neural-network/"
Figure 2.1 "http://www.ai-junkie.com/ann/som/som3.html"
Figure 2.2 "http://www.ai-junkie.com/ann/som/som4.html"
Figure 2.3 "www.cs.umb.edu/~marc/cs672/net11-23.**ppt**"
Figure 2.4 "www.cs.umb.edu/~marc/cs672/net11-23.**ppt**"
Figure 3.1 Screenshot from project
Figure 3.2 Screenshot from project
Figure 3.3 Screenshot from project
Figure 3.4 Screenshot from project
Figure 3.5 Screenshot from project
Figure 3.6 Screenshot from project
Figure 3.7 Screenshot from project
Figure 3.8 Screenshot from project
Figure 3.9 Screenshot from project

Appendix C---Minutes

Minutes for formal meetings can be found through this link:

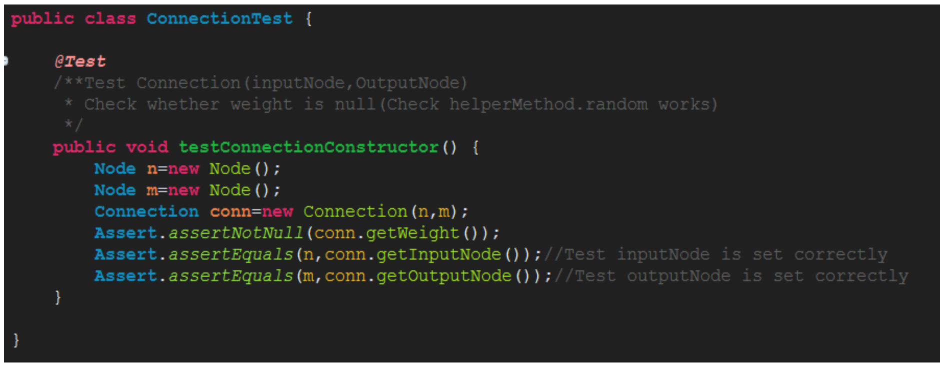
https://code.cs.nott.ac.uk/p/gp14-jl-rij/doc/

Appendix D---JUnit Testing

1.Common test and normalisation tests

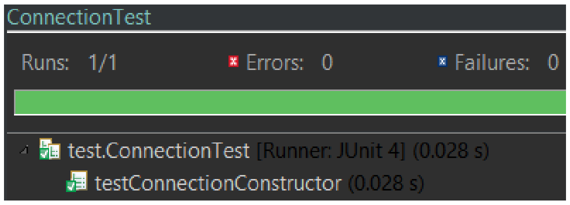
The test in Figure 1.1 is used to test whether a connection is constructed properly. It is tested by checking that the input node and output node are as expected.

The result is shown below in figure 1.2 and it works as expected.



```
public class ConnectionTest {

    @Test
    /**Test Connection(inputNode,OutputNode)
     * Check whether weight is null(Check helperMethod.random works)
     */
    public void testConnectionConstructor() {
        Node n=new Node();
        Node m=new Node();
        Connection conn=new Connection(n,m);
        Assert.assertNotNull(conn.getWeight());
        Assert.assertEquals(n,conn.getInputNode());//Test inputNode is set correctly
        Assert.assertEquals(m,conn.getOutputNode());//Test outputNode is set correctly
    }

}
```

Figure 1.1-Connection Constructor Test

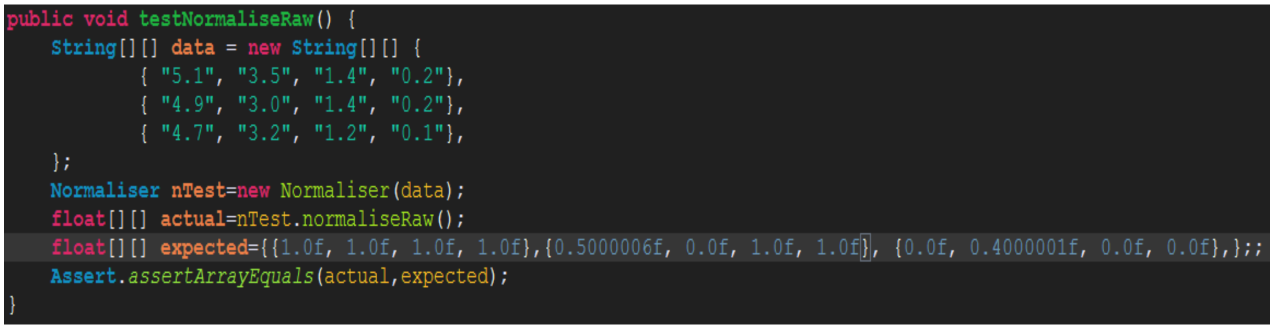


```
ConnectionTest
Runs:  1/1          Errors:  0          Failures:  0

test.ConnectionTest [Runner: JUnit 4] (0.028 s)
    testConnectionConstructor (0.028 s)
```

Figure 1.2-Result of Connection Constructor Test

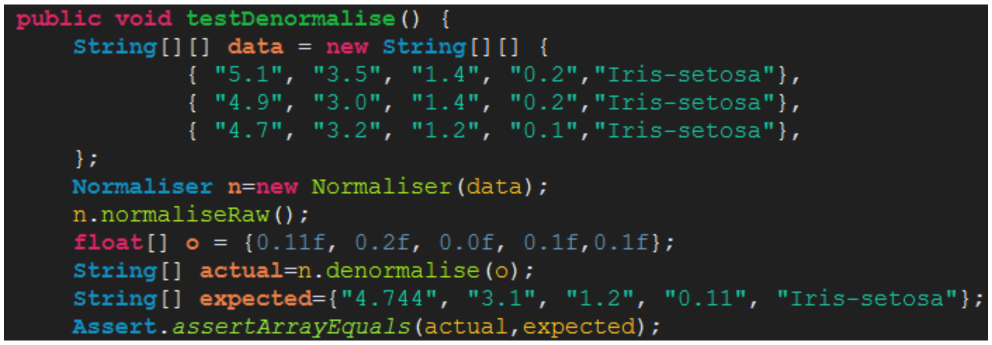
The test in Figure 1.3 is used to test that the normalisation of the data is performed properly by creating an assertion to check the expected value is equal to the actual value.

The test in Figure 1.4 is used to perform tests on the de-normalisation class. This is done with an assertion to check actual value compared to the actual value.



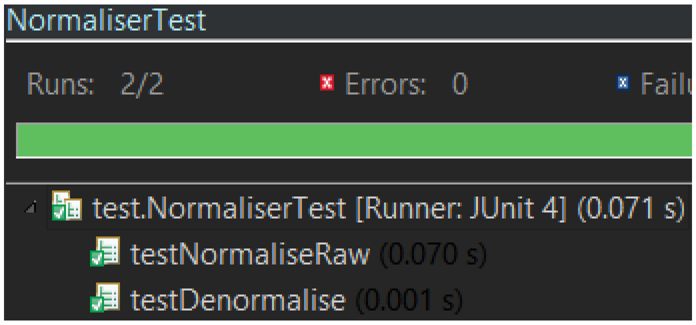
```java
public void testNormaliseRaw() {
    String[][] data = new String[][] {
            { "5.1", "3.5", "1.4", "0.2"},
            { "4.9", "3.0", "1.4", "0.2"},
            { "4.7", "3.2", "1.2", "0.1"},
    };
    Normaliser nTest=new Normaliser(data);
    float[][] actual=nTest.normaliseRaw();
    float[][] expected={{1.0f, 1.0f, 1.0f, 1.0f},{0.5000006f, 0.0f, 1.0f, 1.0f}, {0.0f, 0.4000001f, 0.0f, 0.0f},};;
    Assert.assertArrayEquals(actual,expected);
}
```

Figure 1.3



```java
public void testDenormalise() {
    String[][] data = new String[][] {
            { "5.1", "3.5", "1.4", "0.2","Iris-setosa"},
            { "4.9", "3.0", "1.4", "0.2","Iris-setosa"},
            { "4.7", "3.2", "1.2", "0.1","Iris-setosa"},
    };
    Normaliser n=new Normaliser(data);
    n.normaliseRaw();
    float[] o = {0.11f, 0.2f, 0.0f, 0.1f,0.1f};
    String[] actual=n.denormalise(o);
    String[] expected={"4.744", "3.1", "1.2", "0.11", "Iris-setosa"};
    Assert.assertArrayEquals(actual,expected);
```

Figure 1.4



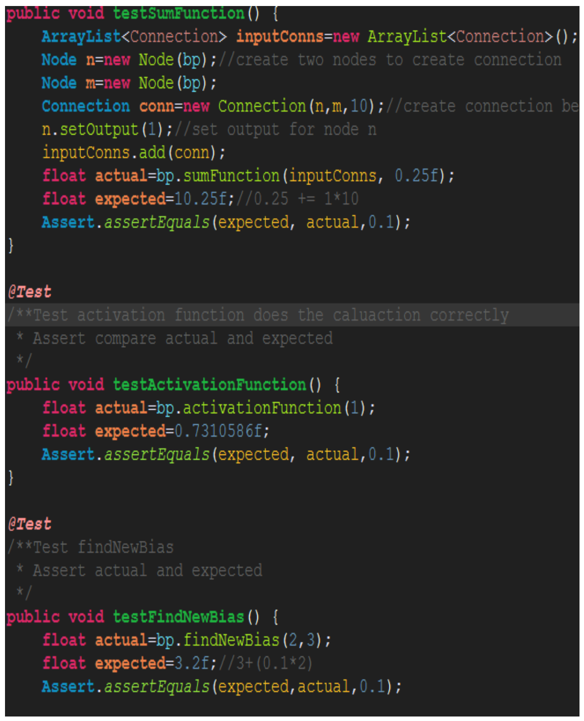
```
NormaliserTest
Runs:   2/2          Errors:   0          Failu
test.NormaliserTest [Runner: JUnit 4] (0.071 s)
    testNormaliseRaw (0.070 s)
    testDenormalise (0.001 s)
```

Figure 1.5

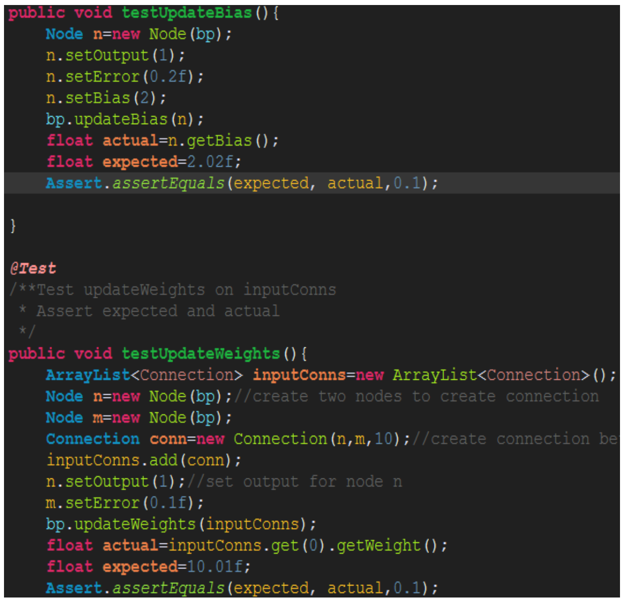
The results can be seen in Figure 1.5 for both normalising and de-normalising . All worked as expected.

2.Backpropagation test

The Following pictures are used to test the backpropagation by testing all the vital calculation classes.



```java
public void testSumFunction() {
    ArrayList<Connection> inputConns=new ArrayList<Connection>();
    Node n=new Node(bp);//create two nodes to create connection
    Node m=new Node(bp);
    Connection conn=new Connection(n,m,10);//create connection be
    n.setOutput(1);//set output for node n
    inputConns.add(conn);
    float actual=bp.sumFunction(inputConns, 0.25f);
    float expected=10.25f;//0.25 += 1*10
    Assert.assertEquals(expected, actual,0.1);
}

@Test
/**Test activation function does the caluaction correctly
 * Assert compare actual and expected
 */
public void testActivationFunction() {
    float actual=bp.activationFunction(1);
    float expected=0.7310586f;
    Assert.assertEquals(expected, actual,0.1);
}

@Test
/**Test findNewBias
 * Assert actual and expected
 */
public void testFindNewBias() {
    float actual=bp.findNewBias(2,3);
    float expected=3.2f;//3+(0.1*2)
    Assert.assertEquals(expected,actual,0.1);
```



```java
public void testUpdateBias(){
    Node n=new Node(bp);
    n.setOutput(1);
    n.setError(0.2f);
    n.setBias(2);
    bp.updateBias(n);
    float actual=n.getBias();
    float expected=2.02f;
    Assert.assertEquals(expected, actual,0.1);

}

@Test
/**Test updateWeights on inputConns
 * Assert expected and actual
 */
public void testUpdateWeights(){
    ArrayList<Connection> inputConns=new ArrayList<Connection>();
    Node n=new Node(bp);//create two nodes to create connection
    Node m=new Node(bp);
    Connection conn=new Connection(n,m,10);//create connection be
    inputConns.add(conn);
    n.setOutput(1);//set output for node n
    m.setError(0.1f);
    bp.updateWeights(inputConns);
    float actual=inputConns.get(0).getWeight();
    float expected=10.01f;
    Assert.assertEquals(expected, actual,0.1);
```

Figure 2.1

Figure 2.2

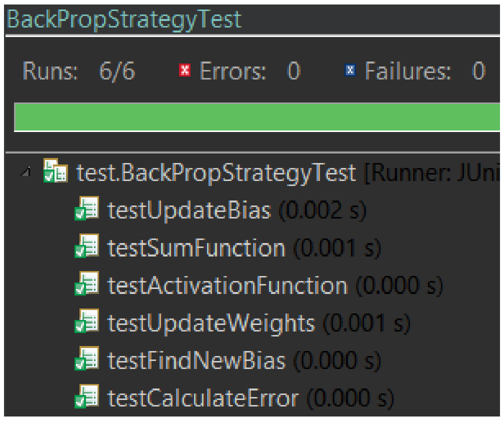

```
BackPropStrategyTest
Runs: 6/6    Errors: 0    Failures: 0

test.BackPropStrategyTest [Runner: JUni
    testUpdateBias (0.002 s)
    testSumFunction (0.001 s)
    testActivationFunction (0.000 s)
    testUpdateWeights (0.001 s)
    testFindNewBias (0.000 s)
    testCalculateError (0.000 s)
```

Figure 2.3

The tests passed as seen in figure 2.3.

3.Rbf test

Not many classes could be tested in junit because of private methods. But the two tests can be seen below:

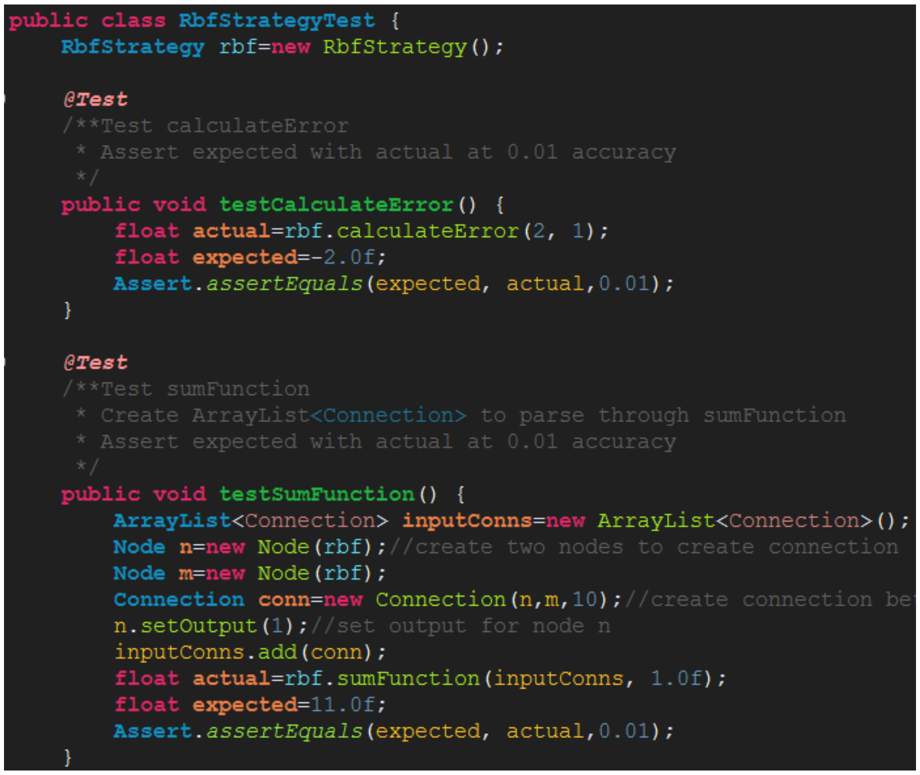
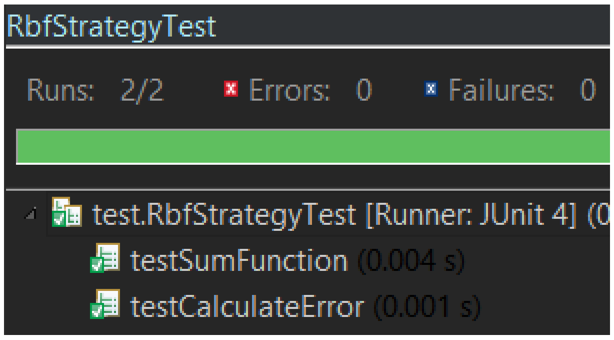

```
public class RbfStrategyTest {
    RbfStrategy rbf=new RbfStrategy();

    @Test
    /**Test calculateError
     * Assert expected with actual at 0.01 accuracy
     */
    public void testCalculateError() {
        float actual=rbf.calculateError(2, 1);
        float expected=-2.0f;
        Assert.assertEquals(expected, actual,0.01);
    }

    @Test
    /**Test sumFunction
     * Create ArrayList<Connection> to parse through sumFunction
     * Assert expected with actual at 0.01 accuracy
     */
    public void testSumFunction() {
        ArrayList<Connection> inputConns=new ArrayList<Connection>();
        Node n=new Node(rbf);//create two nodes to create connection
        Node m=new Node(rbf);
        Connection conn=new Connection(n,m,10);//create connection be
        n.setOutput(1);//set output for node n
        inputConns.add(conn);
        float actual=rbf.sumFunction(inputConns, 1.0f);
        float expected=11.0f;
        Assert.assertEquals(expected, actual,0.01);
    }
```

Figure 3.1

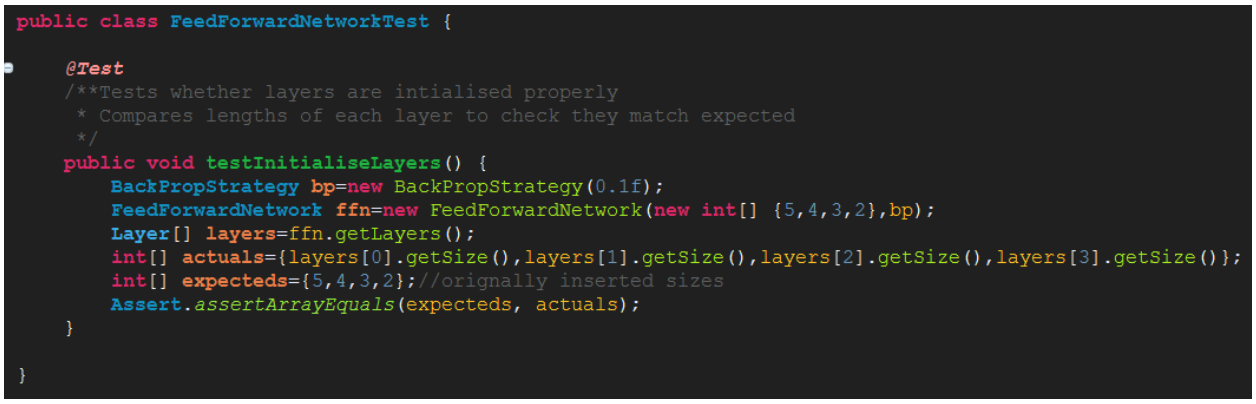


```
RbfStrategyTest
Runs:   2/2      Errors:   0      Failures:   0
test.RbfStrategyTest [Runner: JUnit 4] (0
    testSumFunction (0.004 s)
    testCalculateError (0.001 s)
```

Figure 3.2

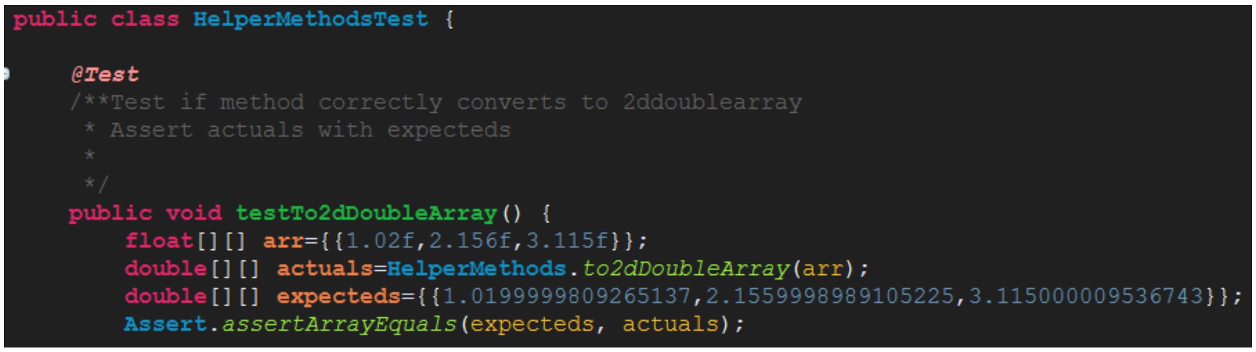
4. Feedforward Network test and helper methods test

In Figure 4.1 and 4.2 test cases can be seen which test whether layers are initialized properly and to test if the array is being converted properly into a double array.



```java
public class FeedForwardNetworkTest {

    @Test
    /**Tests whether layers are intialised properly
     * Compares lengths of each layer to check they match expected
     */
    public void testInitialiseLayers() {
        BackPropStrategy bp=new BackPropStrategy(0.1f);
        FeedForwardNetwork ffn=new FeedForwardNetwork(new int[] {5,4,3,2},bp);
        Layer[] layers=ffn.getLayers();
        int[] actuals={layers[0].getSize(),layers[1].getSize(),layers[2].getSize(),layers[3].getSize()};
        int[] expecteds={5,4,3,2};//orignally inserted sizes
        Assert.assertArrayEquals(expecteds, actuals);
    }

}
```

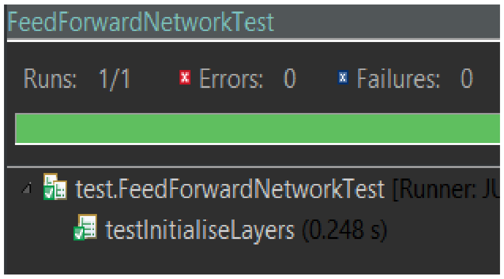
Figure 4.1



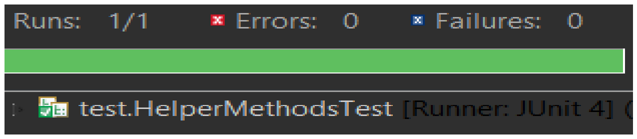
```java
public class HelperMethodsTest {

    @Test
    /**Test if method correctly converts to 2ddoublearray
     * Assert actuals with expecteds
     *
     */
    public void testTo2dDoubleArray() {
        float[][] arr={{1.02f,2.156f,3.115f}};
        double[][] actuals=HelperMethods.to2dDoubleArray(arr);
        double[][] expecteds={{1.0199999809265137,2.1559998989105225,3.115000009536743}};
        Assert.assertArrayEquals(expecteds, actuals);
```

Figure 4.2

All test worked as seen in Figure 4.3 and 4.4



```
FeedForwardNetworkTest
Runs: 1/1    Errors: 0    Failures: 0
test.FeedForwardNetworkTest [Runner: JU
    testInitialiseLayers (0.248 s)
```



```
Runs: 1/1    Errors: 0    Failures: 0
test.HelperMethodsTest [Runner: JUnit 4] (
```

Figure 4.3

Figure 4.4

Appendix E

<u>Functional Requirements</u>

1. The application must have a tutorial for new users, which means interactive, a step-by-step guide on the start screen.
2. The application must have a help function on each screen of the application, which displays a clear and introductory guide for that screen in case of they forget how to use the application.
3. An import tool must be implemented to allow the user to add new data from an external file (csv, txt formats).
4. The import tool must have the ability to choose the input and expected output data from the imported data; it must have example data if no data is imported.
5. The application must have different options depending on the user's experience with neural networks.
6. The application must have an advanced setup mode and a simple setup mode.

a. For advanced setup, users have an option to choose the training algorithm, the number of hidden layers and neural network architectures.

b. For simple setup, depending on the size of the input and output the application must automatically display suggested algorithms.

7. The application must have the ability to change the setup of the neural network at any stage.

a. Must have the ability to change the file at import data stage.

b. Must have the ability to change the neural network at the trained neural network stage.

8. Users have the ability to save and continue on any neural network, which means they can save a trained neural network, or load any previously saved neural network at the initial startup screen.

Appendix F

Non-functional requirements:

1. The application should be compatible with Windows Operating System mainly but also other systems such as Linux.
2. The application must be easy to use.
3. All operations within the application must be able to run independently of each other.
4. The application must not use unnecessary amounts of storage space.
5. The tutorial algorithm must be easy for the users to understand.
6. All operations of changing a different network, resetting the data or changing the data must not take an extensive amount of time.
7. The appearance of the program should be aesthetically appealing.
8. The application must be secure so that the data is not accessible appealing and simple to anyone but the user.
9. The response time of every user’s selected operation must be optimal.

Appendix G

User Manual:

User manual is not included in the appendix as it has been built into the software in the form of a help button and a tutorial mode.