GENERATING STRUCTURE OF A LEARNING MODEL

AUTHOR : Stephan Dale

SUPERVISOR : Jonathan Shapiro

BSc(Hons) ARTIFICIAL INTELLIGENCE

UNIVERSITY OF MANCHESTER

JUNE 2000

ABSTRACT

This report documents the development of a system to automatically generate the structure of a learning model, specifically a modular neural network.

The system required the development of three sub-systems, these are,

- A system to create, train, and manipulate a neural network
- A system to create, train, and manipulate a modular network
- A system to implement a genetic algorithm, capable of ‘breeding’ neural networks

The systems are combined to produce a program capable of using the genetic algorithm to breed neural network for use in a modular network.

CONTENTS

Page

1. INTRODUCTION

1.1 The Problem Considered

There are many computational systems capable of learning to perform a task. Most of these systems have a fixed structure, and learn by altering various parameters until an internal representation of the problem is built up. However, there is no systematic method for learning the structure of such systems.

The process of manually evaluating a task to determine which structures might be suitable is a time-consuming and complex one, and often requires a very high level of domain knowledge. This project is concerned with automatically generating the structure, without requiring an in-depth knowledge of the task.

Three main systems will combine to produce an overall system:

1. Artificial neural networks will be used as "experts" capable of learning a simple task given suitable input-output pairs.
2. A modular network (sometimes referred to in literature as a "mixture of experts") will govern a set of these experts and learn which are most suited to particular regions of input space. This has the effect of decomposing the task into a set of sub-tasks, each performed by a different expert.
3. Finally, a genetic algorithm will take a population of experts and "breed" them to combine the best properties of existing experts to produce new experts more capable of performing the task.

The combination of a modular network, which produces a decision as to which experts are most suited to a sub-task, and a genetic algorithm, which makes beneficial structural changes to the experts, should allow expert structures to be built.

Its worth noting that a modular network is a type of neural network, and modular networks can themselves be used as experts to produce a hierarchical structure [1].

1.2 Practical Relevance

Neural networks have been shown to be very successful learning models. The range of tasks they can learn to perform is vast, and their ability to generalise makes them particularly impressive. Whilst many of the applications they have been applied to are fairly abstract, they have also been used to perform many popular tasks such as object and speech recognition. But probably the most fundamental use is research into the human brain, on which neural networks are modelled.

The manual design of neural networks a laborious and complex process. If the design can be achieved quickly by an automated process, then the time saved can be put to better use evaluating the structures that have emerged. Also, the solution to tasks that cannot currently be performed may become possible. The evaluation of the structures generated would in turn lead to a greater understanding of those tasks.

1.3 Previous Work

1.3.1 Neural Networks

Neural networks are very common in the field of artificial intelligence. A range of variations of both the structure and training methods has been devised, but the general principles remain unchanged. Material on neural networks can be found in a great deal of artificial intelligence literature, such as [7] or [8].

Modular networks are a more recent development of neural networks, see Jacobs, Jordan, Nowlan, & Hinton (1991) [3]. Since this is a relatively new idea only a few variations have been developed. The most substantial of these are the use of a localised gating for the modular network, Ramamurti & Ghosh [9], and the EM training algorithm devised by Jacobs, and Jordan (1993) [4].

The neural networks I will be using are standard, and have neither a structure nor use training techniques other than those originally developed.

1.3.2 Genetic Algorithms

Genetic algorithms are also very common. The basic algorithm is the same for most of these systems [7]. A genetic algorithm uses a number of other algorithms. An example is the mutation algorithm that is responsible for making small random changes to the population. Many implementations of these algorithms exist.

There is a difference between existing genetic algorithms and the algorithm used in this project. While most genetic algorithms encode the population members into bit-strings before operating on them, my algorithm works directly on the expert networks, without encoding them first.

1.3.3 Structure Generation

There exist systems that can build up the structure of neural networks. Two main techniques are used - both rely on some form of genetic algorithm. The first method applies only mutation. This will alter the structure of the networks, such as adding or removing nodes, to attempt to produce a better network. The mutations that are applied can either be randomly chosen, or are determined by a method to predict which will produce the best network [6]. The second method uses a genetic algorithm that implements some form of crossover as well as mutation. This both combines the structural properties of existing networks in the population, and applies mutation, to create new networks. The mutation is nearly always chosen using a random process.

There also exist systems to build the structure of modular networks. The techniques alter the number of experts in the modular network. There are two ways this can happen. Either, a small initial modular network is created and then grown by adding experts. Or, a large initial network is created and shrunk, by removing experts. The network size will continue to be altered until an optimal size is found [9], [10].

I know of no techniques that use genetic algorithms to breed the experts for a modular network.

1.4 Objectives

There are four stages to this project. The three systems described above (1.1) must be coded. These systems must then be combined to make an overall system capable of generating the structure of experts, and hence the structure of the modular network. There is a specific order to these stages:

1. The code to instantiate and train a neural (expert) network must be written. Once this is completed, they can be used as experts in a modular network.
2. It seems a natural progression to write the code to instantiate and train a modular network next.
3. The genetic algorithm can be coded.
4. Finally, the three previous sub-systems must be combined to produce an overall system capable of automatically generating the structure. I will refer to this system as the "modular breeder".

The bulk of the work will be in the first three stages. The fourth stage will call methods defined on the previous three sub-systems, to produce the overall system. Most of the computational work will be done by the three sub-systems.

1.5 Working Environment

I primarily used machines running Linux, with the Java Development Kit (jdk). A number of versions of jdk are available on the university machines. The most recent is version 2.0. My code is known to be compatible with versions 2.0, 1.1.7, and 1.1.8. However, the graphical interface requires Swing version 1.1, rather than version 1.0 which is provided on the university computers. Downloading the required classes from Sun's website and appending the location of the classes to my classpath easily solved this.

My program can write information to files, such as the error of the system as it is trained on a given task. The information is written in a format that is compatible with MATLAB. This allows MATLAB to be used to read the files and generate graphs.

1.6 Outline of Remaining Report

The remainder of the report is organised as follows. Section 2 presents the theory behind the three sub-systems, and describes their combination into the complete system. Section 3 illustrates the design and implementation of the system. This includes the requirements of the system, the choice of language, and the structure of the program. Section 4 introduces the experiments that were carried out to test the system, why they were chosen, and presents various results that demonstrate the system's current performance. Finally, section 5 gives conclusions of the project, including the achievements, possible improvements and extensions.

2. THEORY

2.1 Expert Network

2.1.1 Architecture

I have implemented a feed-forward, layered, neural network, as described by Pratt [8]. This type of network consists of a number of inputs, outputs, and hidden nodes, organised into layers. The nodes only connect to nodes in the layer below. Figure 2.1 shows a network with an input layer, an output layer, and two hidden layers. The nodes have full connectivity (with only the leftmost connections shown for clarity).

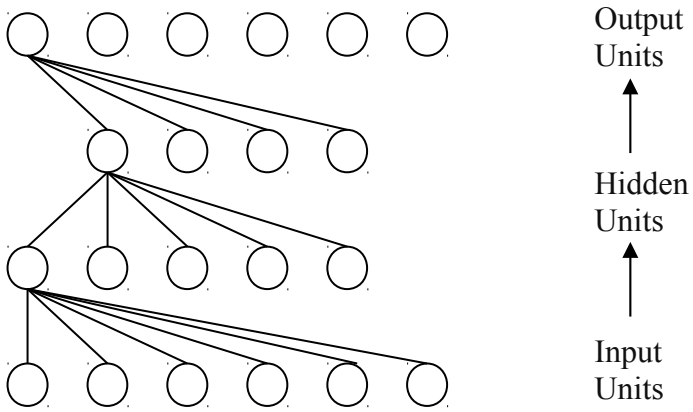
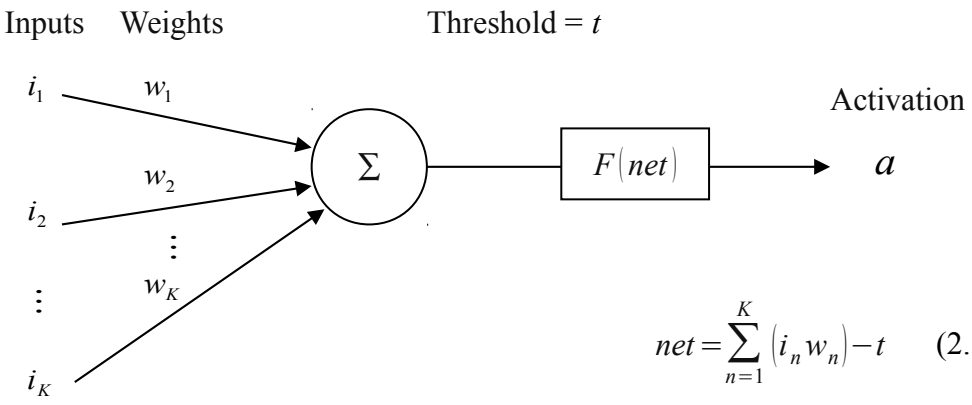


Figure 2.1: Feed-forward, layered, network architecture

Each node has associated with it a number of inputs, a set of weights corresponding to those inputs, and a threshold. The output of the node is calculated by a function of the net weighted input into the node, as shown in figure 2.2.



$$net = \sum_{n=1}^{K} \left(i_n w_n \right) - t \qquad (2.1)$$

$$a = F\left(net \right) \qquad (2.2)$$

Figure 2.2: Node activation.

The node function will be the Sigmoid function,

$$F(x)=\frac{1}{1+e^{-x/T}} \tag{2.3}$$

where T is a small positive constant. This is a standard function used in neural networks, and a description of it can be found in many texts, such as [7] or [8].

2.1.2 Training

The network will be trained using the Backpropagation algorithm. This is as described by Pratt [8]. This algorithm works as follows. First the inputs into the network are set, and the activations of the nodes allowed to propagate through the network to produce an output. The errors of the output nodes are then calculated using,

$$\delta_j^p=\left(d_j^p-a_j^p\right) \tag{2.4}$$

where δ_j^p is the error of (output) node j given pattern p, d_j^p is the desired output of the node, and a_j^p is the actual output of the node.
These errors are then propagated back through the network by being used to calculate the errors of the hidden nodes,

$$\delta_i^p=\sum_j w_{j,i}\delta_j^p \tag{2.5}$$

where $w_{j,i}$ is the weight of the connection from node j to node i.
The weights of all nodes are adjusted using,

$$w_{j,i}:=w_{j,i}+\eta a_i^p \delta_j^p f'\left(net_j^p\right) \tag{2.6}$$

where η is a small positive constant, and $f'\left(net_j^p\right)$ is the differentiated function acting on the net output of node j given pattern p.
Finally, the thresholds of the nodes are adjusted using,

$$\theta_j:=\theta_j-\eta\delta_j^p f'\left(net_j^p\right) \tag{2.7}$$

where θ_j is the threshold of node j.

As this algorithm requires the differentiation of the function in order to calculate the weight and threshold changes, it becomes inconvenient if different functions are used in different nodes. To allow this, if no differentiated function is supplied to the Backpropagation method, it becomes,

$$f'(net)=1.0 \tag{2.8}$$

2.1.3 Structural Operations

In order to breed networks, a number of methods were implemented to allow structural alteration. These methods enable the following,

1. Addition/removal of a layer.
2. Addition/removal of a node.
3. Addition/removal of a connection.
4. Setting of a node's connection weights.
5. Setting of a node's function.
6. Setting of a node's connectivity pattern.
7. Setting of a node's threshold.
8. Activation/deactivation of a node.

There is also a range of methods that return the structural parameters of the network.

2.2 Modular Network

2.2.1 Architecture

The modular network implemented is described by Haykin [1] (also see [2]). It consists of a number of experts that are governed by a gating network. Given an input pattern, the gating network produces a decision as to which experts are most capable of producing a correct output. A weight is produced corresponding to each expert, which scales its output according to its ability. As mentioned above, the experts are neural networks, although they could be any learning model. The architecture of the modular network is shown below, in figure 2.3.

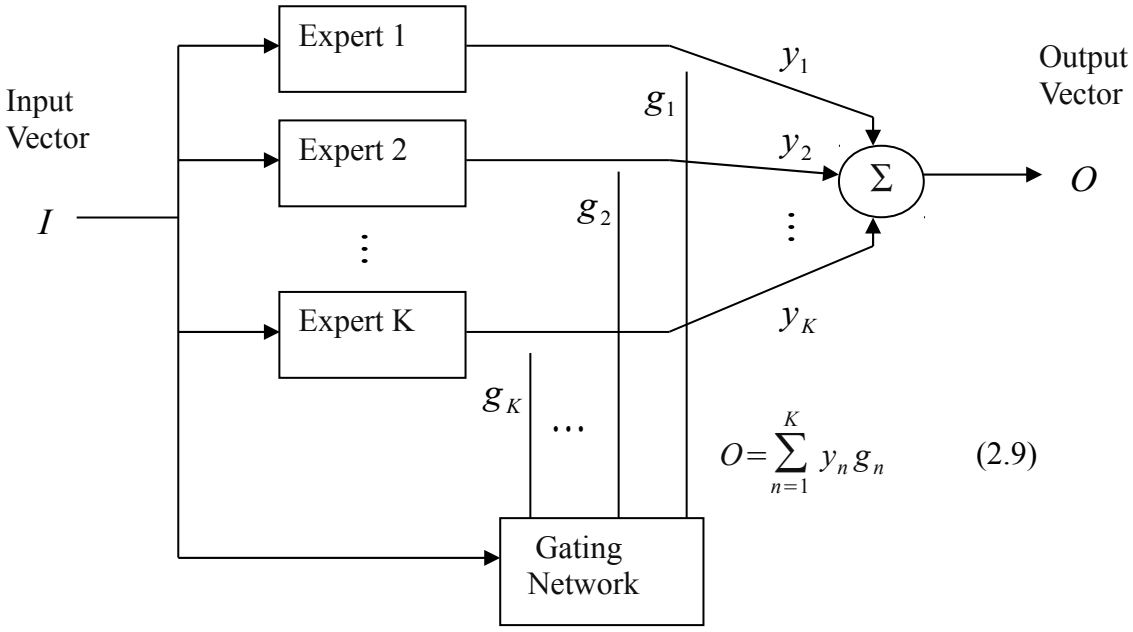


Figure 2.3: Modular network architecture

The gating network consists of a number of so-called “Softmax” nodes. This is due to the function they use, called a Softmax function. This was devised in 1990 by Bridle. The network consists of a single layer of softmax nodes fully connected to the input layer. Figure 2.4 shows this network (with only the connections of the first node shown for clarity).

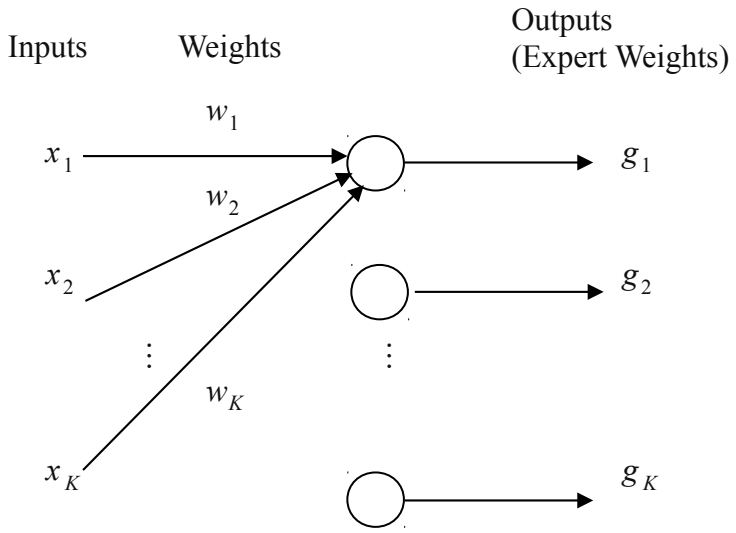


Figure 2.4: The gating network

Each of these nodes corresponds to an expert, and produces the weight for that expert.

The Softmax function is,

$$g_k = \frac{e^{u_k}}{\sum_{n=1}^{K} e^{u_n}}, \qquad k = 1,2,\cdots,K \qquad (2.10)$$

where u_k is the weighted input into node k,

$$u_k = w_k x_k, \qquad k = 1,2,\cdots,K \qquad (2.11)$$

2.2.2 Training

Two algorithms have been implemented. The first is very simple and will not be explained here. The second is based on the Winner-takes-all procedure, described by Jacobs, Jordan, and Barto [2]. This is a more complex procedure. The algorithm works as follows. First, the modular network is presented with an input pattern. The error of the network is then calculated using,

$$E^p = \frac{1}{2}\left(d^p - a^p\right)\left(d^p - a^p\right) \qquad (2.12)$$

where E^p is the error of the network on pattern p, d^p is the desired output of the network, and a^p is the actual output.

This error is taken to be the current performance of the network. Next, it needs to be determined whether the performance of the network has significantly improved or not. This is done by comparing the networks current performance with the networks past performance on the same input pattern. The past performance of the network is calculated iteratively using,

$$\bar{E}^p(t) = \alpha E^p(t) + (1-\alpha)\bar{E}^p(t-1) \tag{2.13}$$

where $\bar{E}^p(t)$ is the measure of past performance at time step t on pattern p, and α is a constant, $0<\alpha<1$, that determines how rapidly past values of E^p are forgotten.

The comparison that evaluates true if the network has improved is,

$$E^p(t) < \gamma \bar{E}^p(t-1) \tag{2.14}$$

where γ is a factor that determines how much the performance must have increased for the network to be considered improved.

The errors of the experts then need to be determined by calculating their error using,

$$E_i^p = \left| \frac{\sum_{j=1}^{N} \delta_j^p}{N} \right| \tag{2.15}$$

where E_k^p is the error of expert k on pattern p, δ_j^p is the error of output j (as calculated in the Backpropagation algorithm described in section 2.1.2), and N is the total number of outputs.

The expert with the least error is taken to be the winner, and all others are taken to be the losers. If the network has improved, then the desired output of the softmax node corresponding to the winning expert is set to 1, and the desired outputs of the softmax nodes corresponding to the all other experts are set to 0. The weights of the softmax nodes are then adjusted to bring the output of the nodes closer to their desired outputs, using

$$w_k := w_k + \eta x_k \delta_k \qquad k = 1,2,\cdots,K \tag{2.16}$$

where δ_k is calculated in the same way as the experts. On the other hand, if the network has not improved, then the desired outputs are all set to bring the outputs of the nodes closer to a neutral value.

This neutral value is $\frac{1}{N}$, where N is the number of experts.

Finally, the desired outputs of the experts are adjusted proportionally to the weight produced by the corresponding softmax nodes. This is to influence the training of each expert according to it's performance. The desired outputs are adjusted as

follows,

$$\bar{d}_k^p = g_k^p\left(d_k^p - a_k^p\right) + a_k^p \tag{2.17}$$

where $\bar{d}_k^p$ is the altered desired output of expert k on pattern p.

For a discussion of the effects of this algorithm, please see [2].

2.3 Genetic Algorithm

2.3.1 General Method

The general method is described by Mitchell [7]. The following description is as implemented in this project. You start with an initial population of neural networks. The method describes the program cycle that creates new populations of networks. The cycle is simple, as the majority of work is done by four algorithms employed by the general method. These algorithms are as follows,

1. Fitness evaluator: This evaluates the fitness of each population member.
2. Selection algorithm: This selects a member from the population. The fitness of the member is commonly used to determine selection.
3. Crossover operation: This takes a set of population members, called the "parents", and produces "child" members, consisting of various properties taken from the parents.
4. Mutation algorithm: This takes a population member and applies some kind of mutation to its properties.

The networks in the initial population are initialised to have a random number of layers, and a random number of nodes per layer, both parameters set by the user.

The program cycle is as follows,

1. Use the fitness evaluator to assign a fitness to each member in P, where P is the population
2. Probabilistically select $\frac{N}{100}(100-c)$ members of P to add to P_s, using the selection algorithm, where N is the size of P, c is the percentage of population members to crossover, and P_s is the new population.
3. Probabilistically select $\frac{N}{100}c$ members of P, using the selection algorithm, to be used as parents, and produce children using the crossover operation.
4. Add these children to P_s.
5. Mutate m percent of the members of P_s, using the mutation algorithm.
6. Update $P \leftarrow P_s$.
7. Repeat from step 1, until desired fitness has been reached.

As previously mentioned, the four algorithms described above perform the bulk of

work. They are described below.

2.3.2 Fitness Evaluator

Two algorithms have been produced. They are both based on the same general method. For both algorithms, the error of the population member, E, is calculated as follows,

$$E = \sum_{p=1}^{K} \left(d^p - a^p \right) \tag{2.18}$$

The inverse of this error is then taken to produce a fitness measure, F,

$$F = \frac{1}{E} \tag{2.19}$$

In the first algorithm, this is the fitness measure assigned to the population member. The second algorithm assigns the square of this value as the members fitness,

$$F' = F^2 \tag{2.20}$$

where F' is the fitness this algorithm assigns to the member.

2.3.3 Selection Algorithm

The selection algorithm is fitness proportionate. It works as follows. A member, i, is randomly selected from the population. The probability of selection, pr, is then calculated using,

$$pr = \frac{F^i}{\sum_{n=1}^{N} F^n} \tag{2.21}$$

where F^i is the fitness of the member.

A random number, α, is generated, $0 < \alpha < 1$, and compared with pr. If $pr > \alpha$, then i is selected, otherwise the cycle repeats until a member is selected.

2.3.4 Crossover Operation

The crossover operation takes two parents, and produces two children. It operates by randomly selecting a layer number common to both parents (excluding the input and output layers), and positions in that layer. The parents are then cut, and the halves spliced together to produce the children. This is best illustrated with a diagram, see figure 2.5 below.

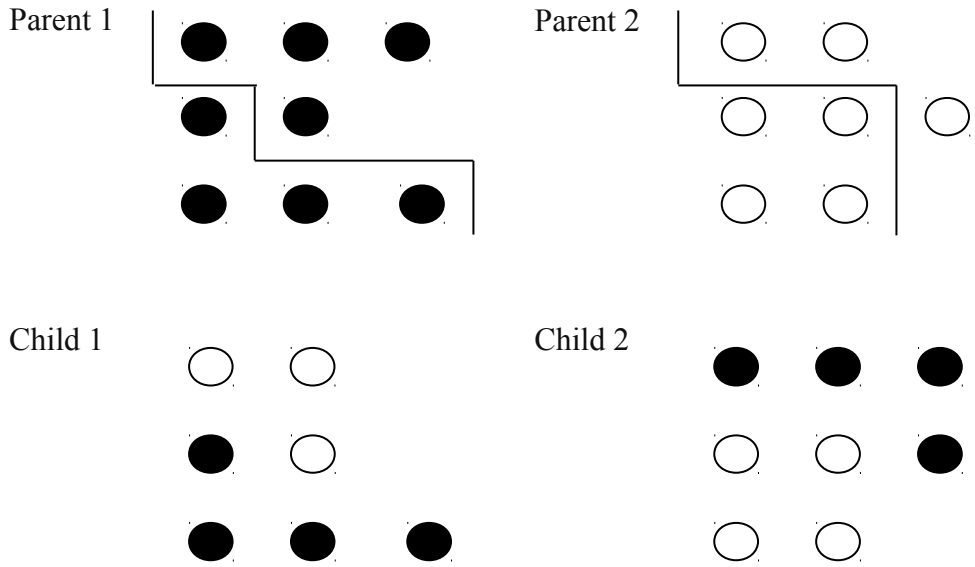


Figure 2.5: An example of the crossover operation

The functions and thresholds of the nodes are kept unchanged. The connections may be altered to ensure that the connectivity pattern is preserved, but any connections that can remain, do so, along with their corresponding weights.

2.3.5 Mutation Algorithm

Five mutation algorithms have been implemented. They work as follows,

1. MutateAll: This has the possibility of applying any of the possible structural alterations. These are addition of a connection, node, or layer, or the removal of a node, or layer. The decision not to implement addition of connections is to prevent multiple connections between nodes. All choices are completely random, and have equal probability.
2. MutateAllButConnections: This carries out the same mutation as MutateAll, except the connections are not mutated.
3. MutateNodes: A node is added or removed from a hidden layer randomly chosen.
4. MutateConnections: This operation removes a connection from one of the nodes, selected randomly. It could be either a hidden node, or an output.
5. NodeDeactivationMutation: This algorithm deactivates nodes at random (except for input and output nodes). There is the danger with this algorithm that all nodes in a layer will be deactivated, leading to a useless network. For this reason, it is best used when dealing with large network, where the possibility of deactivating all nodes in a layer is slim.

2.4 Modular Breeder

2.4.1 Integration Aims

The three systems described above must now be integrated. The aim is to make a system capable of taking a set patterns that define a task, and generate the structure of a modular network capable of performing it. The idea is as follows. The modular network will decompose the task into a set of smaller, easier, sub-tasks. Expert networks will be produced for each of the sub-tasks, using the genetic algorithm. This should be made easier due to the simplification of the task. The networks produced by the genetic algorithm will become experts in a new modular network, allowing the entire task to be performed. This cycle will continue some stopping criteria have been met.

2.4.2 The Algorithm

The stages of the algorithm are as follows.

1. An initial population of modular networks is produced randomly, using parameters supplied by the user.
2. The modular networks are trained for a user-defined number of epochs.
3. For each input pattern, the best experts from each modular network are recorded. The best experts are determined by looking at the weights that the modular network allocates to the experts, given the input pattern.
4. The best experts from the first modular network will be used to define the sub-tasks, as follows. For each of the experts, the set of patterns for which they perform best, are grouped. These groups make up the sub-tasks.
5. For each of the patterns making up a sub-task, the corresponding best experts from the other modular networks are recorded. This makes up a group of experts that are best at the sub-task. See figure 2.6.

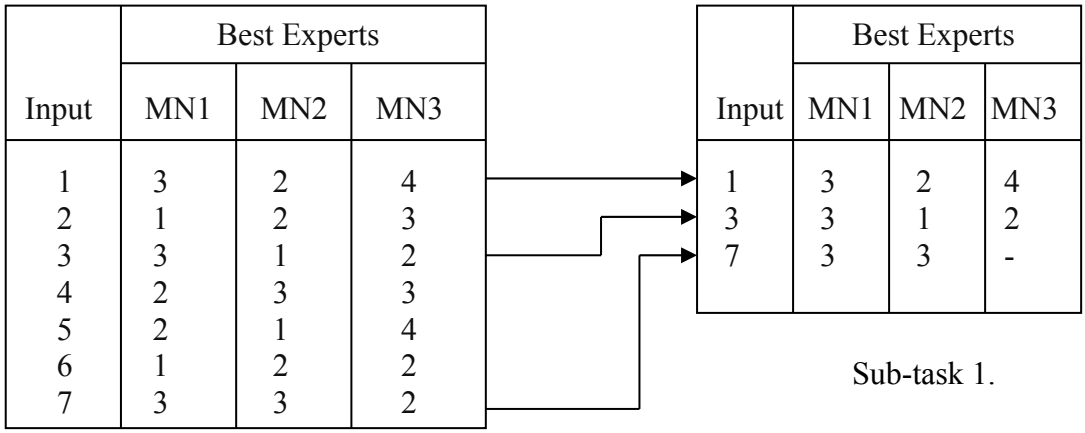


Figure 2.6: An example of sub-task identification (only the first sub-task shown).

6. The groups of best experts for each sub-task will make up separate populations. So you have one population per sub-task.
7. Each population is used in a genetic algorithm. The algorithm will evolve the

experts to produce new populations of experts, each more capable than the last. The patterns used by a particular genetic algorithm are the set of patterns that make up the sub-task for the population. The improvement required by the algorithm is a user-defined parameter.

8. The new populations are then split between the same number of modular networks as in the initial population. The first modular network will receive the best population member from each sub-task population. The second modular network will receive the second best member from each sub-task population, and so on. This creates modular networks with the same number of experts as sub-tasks.
9. The cycle then continues from step2, until the desired error is reached.

Note that the first modular network should be best at the task, as it received the best members from each population to make up its experts. This also means that the first modular network's experts should be most suitable to define the sub-tasks, as they perform the best on them.

3. DESIGN

3.1 Requirements

There are three requirements. The systems must be,

1. Flexible
2. Easy to test
3. Easy to extend

3.1.1 Flexibility

This is the primary requirement. There are three reasons for this.

Firstly, the performance of the systems is dependent on a variety of parameters. Mechanisms must be in place to allow these to be altered, both automatically, and by the user, so that optimal values can be found.

Secondly, the success the systems will experience is unknown. If a method does not perform well, it should be possible to change it for another without needing to re-implement any existing code. This is most relevant for the genetic algorithm, and the modular network breeding system. These use original methods whose success cannot be predicted easily.

Finally, both the structural and behavioural properties of the systems need to be able to be dynamically retrieved, altered, and copied. This is primarily to allow the genetic algorithm, and the modular breeding system, to alter the structure of the networks. It is also relevant to the various training algorithms that need to alter the behaviour of the system.

3.1.2 Ease of Testing

As indicated above, the systems are largely experimental. Testing must be a relatively hastle-free process, allowing the parameters to be changed quickly and easily.

The tests should produce a variety of results that can illustrate properties of the systems. These results must be in a format that can be easily interpreted. The reporting of these results must be able to be switched on and off, allowing different results to be displayed when testing for different properties. This has use in both correctness and performance testing.

3.1.3 Ease of Extending

One of my intentions is to produce a system that can be easily extended. It seems a shame to spend a year developing a system that will never again be put to use. If the proper groundwork can be made on a system that has the potential to be used in the

future, then an effort needs to be made to allow the system to be as extendable as possible.

Another point is that flexibility and ease of extension go hand-in-hand. For a system to be truly flexible, it must also be extendable.

It's worth pointing out that speed is not an issue. I am prepared to put up with a potentially slow program, providing the three requirements listed above are met.

3.2 Choice of Language

This project is implemented in Java. This decision was reached relatively quickly, for several reasons.

Firstly, neural networks and modular networks are inherently object-oriented. They can be decomposed into structural elements that link together in a highly organised way. These elements can easily be represented as objects in Java. Genetic algorithms also seem naturally suited to an object-oriented implementation, as they essentially deal with a population of objects.

Secondly, the integration of the different systems into the final modular network breeding system should be easy. The methods that do the bulk of the work will have already been defined on the sub-system classes. It should just be a matter of instantiating the classes I need, and starting the methods running with the appropriate parameters.

The third reason is because of the need to provide a flexible and extendable program. Java has in-built support for the implementation of different methods (that have a common super-class), and allows them to be dynamically switched between. This allows any new methods to be defined and used without changing any existing code.

Finally, Java is a language that I have quite taken to, as it feels to me like a natural progression from C. For these reasons, it's a language I would like to get more experience of coding in.

3.3 Considerations

In designing the systems to meet the requirements listed above, some considerations come to light.

The first concerns the structure of the code, namely the hierarchical relationships between various classes. This includes the relationships between classes and their super-classes, and between them and the classes they use. Because of the natural decomposition into hierarchies already present in the systems, I don't believe there will be a great deal of ambiguity as to the class hierarchies required.

The second consideration is the identification of the packages that the classes will be

contained in. Again, this will not be a complex process due to the clear identification of the operations performed by the systems implemented.

Progress monitoring of the systems is an issue. A decision has to be made as to what information will be reported in real-time as the systems run, as well as to what information will be needed to analyse the systems performance. This information needs to be stored in such a way that it can be easily interpreted.

The final consideration is that of the interface with the user. A text-based interface will be sufficient to input the parameters, but there is the issue of whether it is easy enough to use, and easy enough to alter parameters.

3.4 Design of the Systems

This section will report the key designs that were used in implementing this project, illustrated with some UML diagrams. The entire project is too substantial to include a complete design. Appendix A provides a full listing of the classes and their methods.

3.4.1 Network class

The abstract class, Network, defines all artificial neural networks. It forces any classes that extend it to implement five methods that it specifies. These methods are the fundamental operations that define an artificial neural network, whether it is a feed-forward, layered network, or a modular network. See figure 3.1 below.

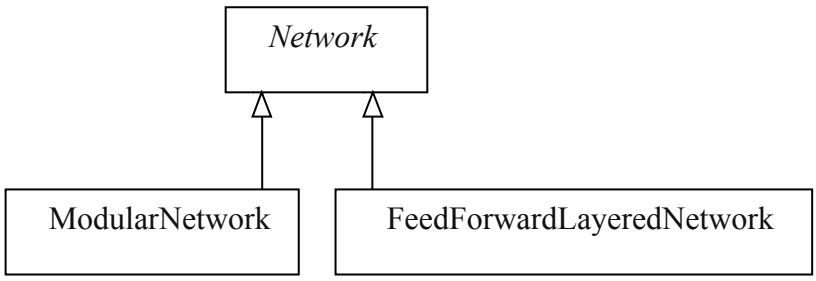


Figure 3.1: The abstract Network class

One example of where this is used is the modular network. The modular network can take any class that extends Network as an expert. This is because the only operations needed for the correct operation of an expert are those that the Network class enforces. This allows the possibility of hierarchical structures, where an expert is in fact another modular network.

3.4.2 Neural Network

The FeedForwardLayeredNetwork class implements the feed-forward, layered, neural network as described in section 2.1. It extends the abstract class Network (figure 3.1). The classes used by FeedForwardLayeredNetwork indicate the structural elements that combine to make a neural network. They are shown below in figure 3.2.

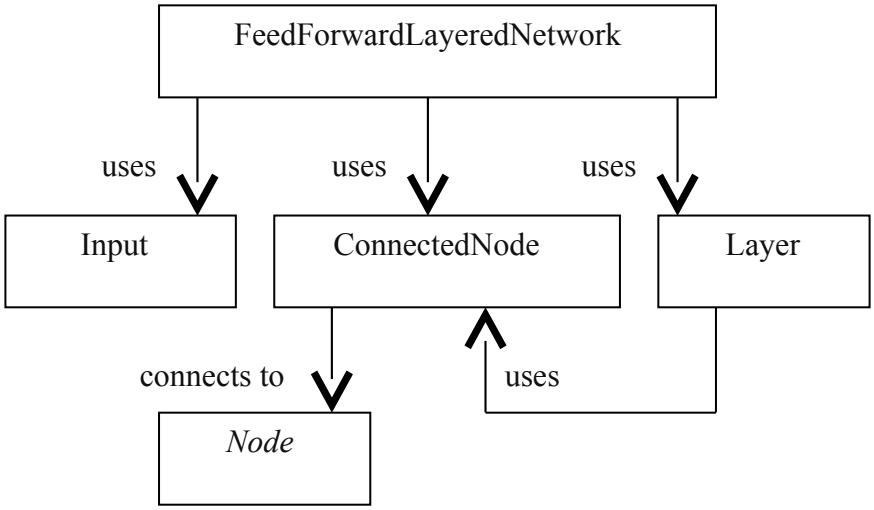


Figure 3.2: The FeedForwardLayeredNetwork class

Class Layer represents the hidden layers that can be present in a neural network. The methods defined on this class take as much work as possible away from FeedForwardLayeredNetwork. Even so, the bulk of the work still has to be done by methods in FeedForwardLayeredNetwork. This is because most of them require manipulation of the rest of the network's components.

Class Input represents an input node. The number of inputs that a FeedForwadLayeredNetwork has does not vary. Because of this, Input objects are stored directly in FeedForwardLayeredNetwork rather than in a Layer - the number of inputs does not change, so there is no need to use the methods provided by Layer. Input nodes do not have connections *to* other nodes, but they do store references to the connections *from* other nodes. The distinction is described shortly.

Class ConnectedNode represents nodes that have connections to other nodes in the network. Most ConnectedNode objects are stored in class Layer. The exceptions to this are the ConnectedNode objects that are used as output nodes. Since the number of outputs in a network will not vary, they can also be stored directly in the FeedForwardLayeredNetwork class, like the Input objects.

Input and ConnectedNode both have a common abstract super-class, Node. Node exists to enforce a set of methods that define a node's properties. The two most important are to set the activation of the node, and get the activation of the node. FeedForwardLayredNetwork calls these two methods on one ConnectedNode at a time, in a particular order. This propagates the node activations through the network

to produce an output.

A ConnectedNode has a threshold, and uses classes Connection, Function, and ConnectivityPattern, as shown in figure 3.3 below.

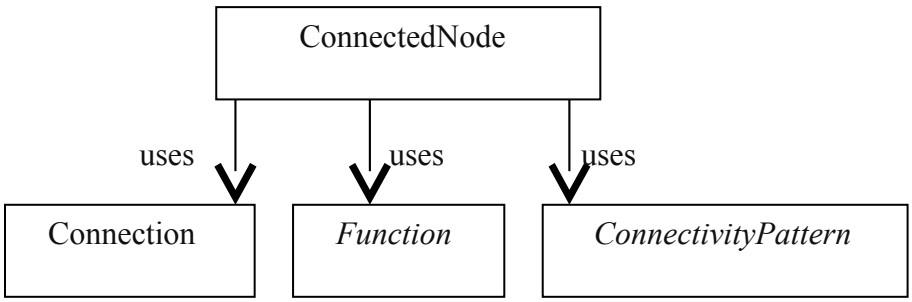


Figure 3.3: The ConnectedNode class

Class Connection implements the connections in the network. A Connection object stores a reference to a Node, and a weight value. It has a method to return the weighted activation of the Node. Nodes store references to any Connections that are from a ConnectedNodes to the Node. This is to allow easy navigation forwards and backwards through the network. See figure 3.4.

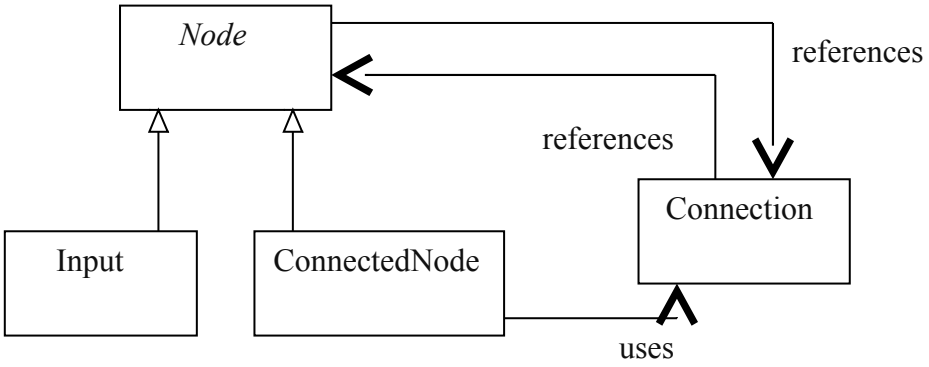


Figure 3.4: The Node class

The Function class is an abstract class that enforces methods to implement a function, such as the Sigmoid function. It takes the ConnectedNode's set of Connections, and produces an activation.

ConnectivityPattern is also an abstract class. It enforces methods that define a connectivity-pattern, given a number of possible Connections. These methods are used by ConnectedNode to ensure that if any methods are called to alter its connections, the connectivity-pattern is upheld. The connectivity-pattern is returned in the form of a bit-string, indicating which Connections will be kept.
The use of these different classes by ConnectedNode, means that a variety of different

functions and connectivity-patterns can be implemented. Some are shown in figure 3.5. This leads to a very flexible neural network implementation.

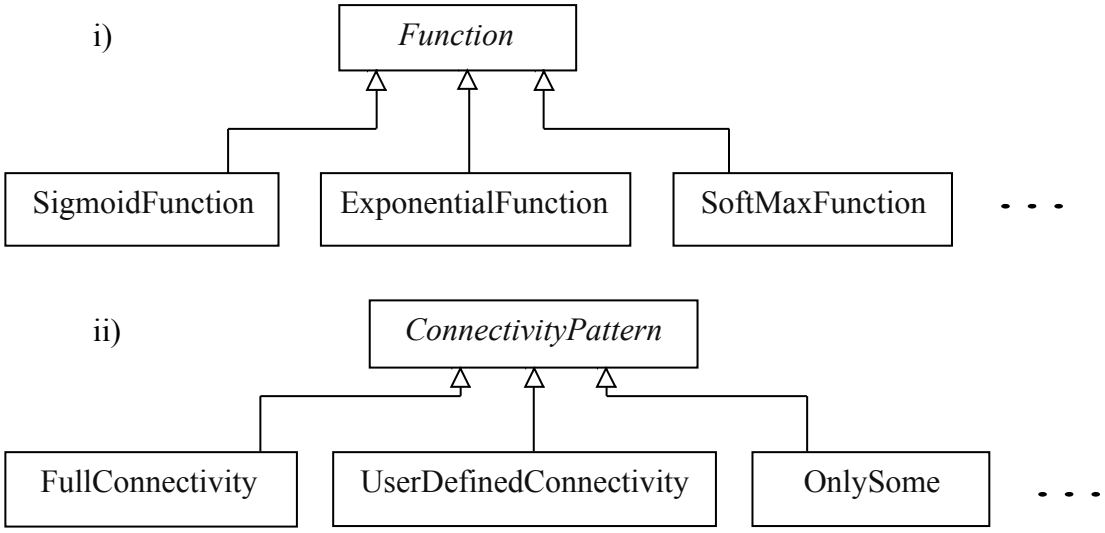


Figure 3.5: Some implementations of Function and ConnectivityPattern

3.4.3 Modular Network

Class ModularNetwork implements a modular network, as described in section 2.2. It extends the abstract class Network. The UML diagram is shown in figure 3.6.

The ModularNetwork class stores a set of Network objects as the experts. The inputs of the modular network are represented by Input objects, as in FeedForwardLayered-Network. The gating network is implemented using ConnectedNodes with softmax functions, and full-connectivity to the Inputs of the modular network.

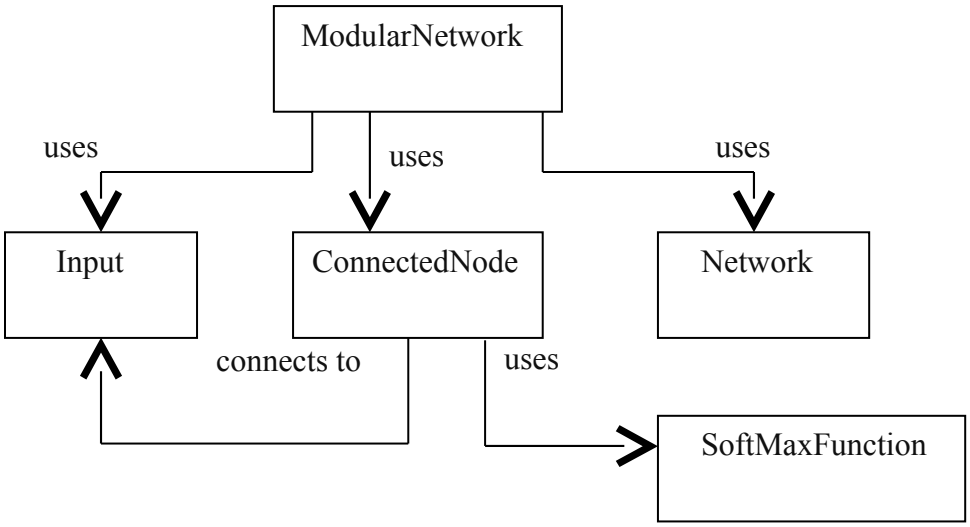


Figure 3.6: The ModularNetwork class

3.4.4 Training of the Neural Network Systems

The training of a FeedForwardLayeredNetwork is performed by a FFLNTraining-Algorithm. This is an abstract class that specifies the train method that a training algorithm must implement. The train method takes a FeedForwardLayeredNetwork, and some input-output patterns. It trains the network using the various methods that FeedForwardLayeredNetwork provides. The class BackPropagation implements the Backpropagation algorithm.

Similarly, the training of a ModularNetwork is performed by a subclass of the abstract class MNTrainingAlgorithm. The train method takes a ModularNetwork, and input-output patterns, and trains it using the methods provided by the ModularNetwork class. There are two MNTrainingAlgorithms, ErrorAdjust, and WinnerTakesAll., shown in figure 3.7.

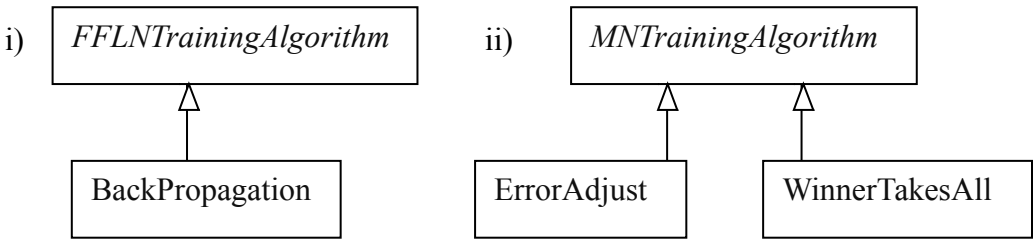


Figure 3.7: The classes that implement the training algorithms

3.4.5 Genetic Algorithm

The genetic algorithm is implemented by class FFLNBreeder. When instantiated, this either takes a population of FeedForwardLayeredNetworks, or parameters that allow a random population to be generated.

There is only one general algorithm for breeding a population of components, which was described in section 2.3. This algorithm is defined by a breed method in FFLNBreeder. To provide a variety of behaviours, FFLNBreeder uses four classes that abstract the fitness evaluator, selection algorithm, crossover operation, and the mutation algorithm. This is shown in figure 3.8 below.

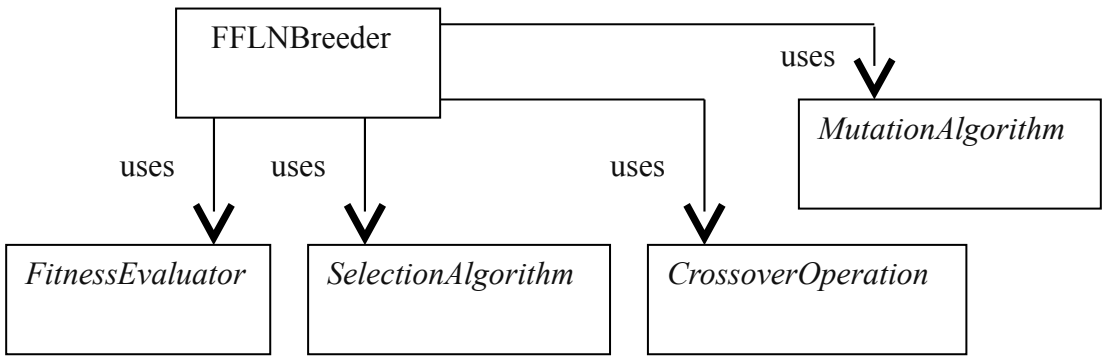


Figure 3.8: The FFLNBreeder class

These classes are extended to provide a range of algorithms. Some of the algorithms implemented are shown below, in figure 3.9.

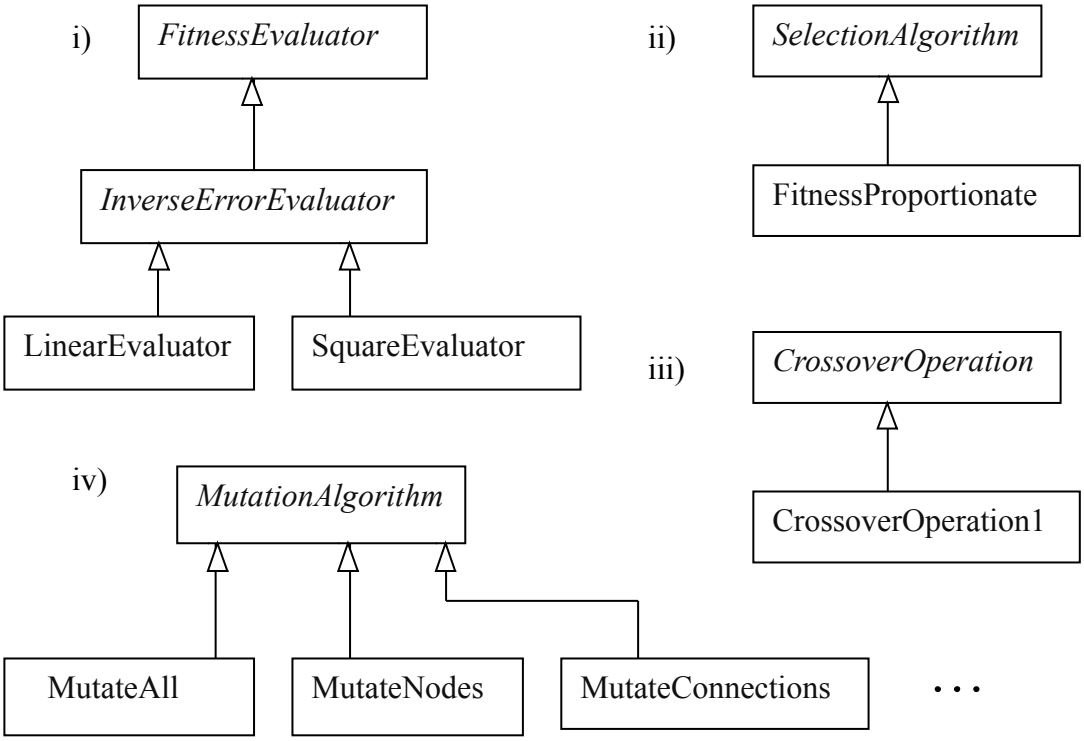


Figure 3.9: Some of the implementations of the classes used by FFLNBreeder

3.4.6 Modular Breeder

The modular network breeding system (as described in section 2.4) is implemented by a class called ModularBreeder. It is instantiated with parameters that allow a random population to be generated. This class uses the three sub-system classes, namely FeedForwardLayeredNetwork, ModularNetwork, and FFLNBreeder, as shown in figure 3.10.

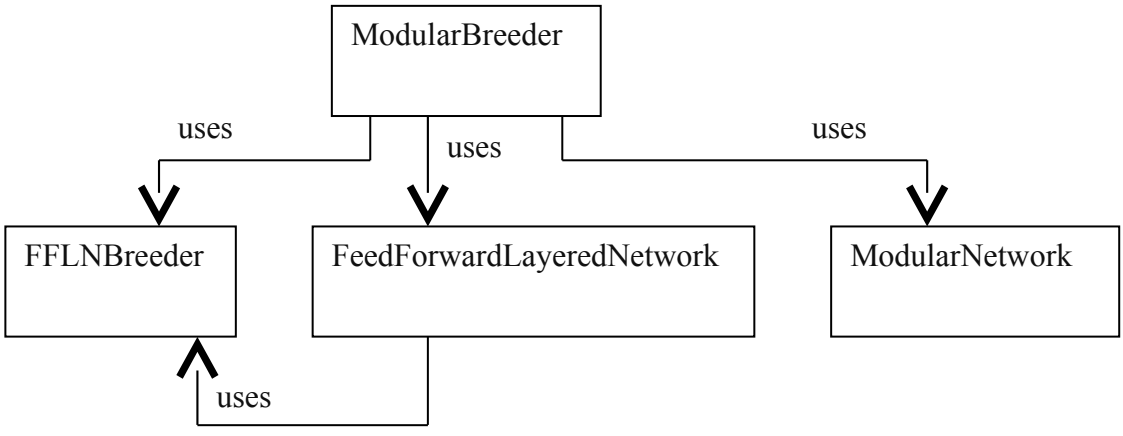


Figure 3.10: The ModularBreeder class

The class defines a method that starts the system's program cycle. The method takes a number of input-output patterns, parameters needed to set up the various sub-systems it uses, as well as parameters relevant only to ModularBreeder (for example, percentage improvement that FFLNBreeder should achieve on each cycle).

3.4.7 Monitors

Monitoring of the systems are carried out by monitor classes. An abstract class called ActivityMonitor is provided as a super-class that all monitor classes should inherit from. It defines a set of methods that can write information to a file. These methods are called by its sub-classes with information needed for the monitoring. Monitoring of a specific property can either be to a given filename, or a general file can be specified for the monitoring of all properties (providing they are switched on).

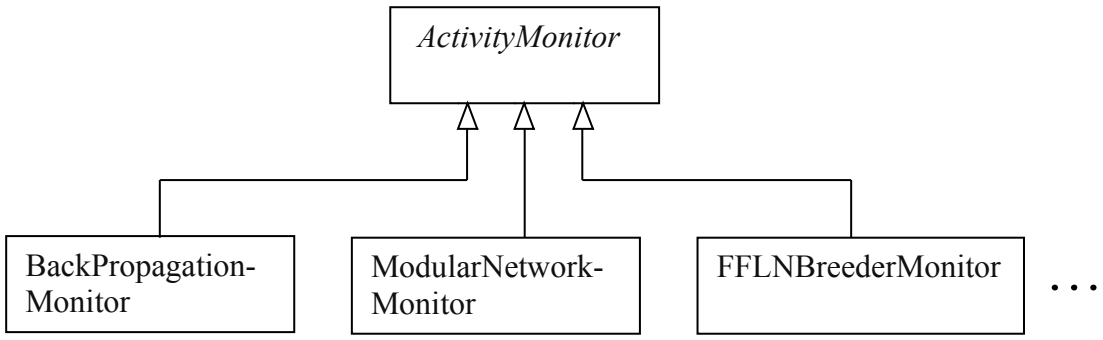


Figure 3.11: The ActivityMonitor class, and some implementations of it

The switching of the monitoring is done in the monitor class, rather than in the class you are monitoring. This is done to localise control to the specific monitor class thus preventing unnecessary code in the class being monitored.

Classes that can be monitored are initialised with a new ActivityMonitor sub-class whenever they are instantiated. The monitor class defines a set of register methods. They take some arguments and write them to a file along with comments, provided a file was previously specified. The class being monitored calls all register methods that are defined on the monitor. The monitor does nothing until one of its methods is called to switch the monitoring of something on.

If a new property is to be monitored, all that is required is a corresponding register method in the monitor class. Calls to the method from the class being monitored will allow monitoring to take place.

3.4.8 Testers

There are a number of classes that I refer to as tester classes. These are pre-compiled classes, written in order to test the correctness of existing classes. This is necessary before they can be extended, or used, by other classes. These tester classes all define

a main method, which is called to run the various tests. These entail instantiating the classes being tested, and calling their methods with a variety of different arguments. The responses of the methods are output to the screen, to allow them to be verified.

3.4.9 Summary of Design

The use of abstract classes is a very powerful mechanism. New classes can be written to extend an abstract class, in order to provide different algorithms and behaviours. No changes are required by any of the classes that use the abstract class. For instance, if a new connectivity-pattern is needed, then a class is created that extends ConnectivityPattern. ConnectedNode (that uses ConnectivityPattern) does not have to change, and hence neither does FeedForwardLayeredNetwork. This applies to all of the abstract classes, and the classes that use them.

Abstract classes were written for all operations where there is a range of possible implementations. This serves to enforce a specification that the algorithms have to adhere to. It provides a framework for implementing new algorithms in the future.

The use of an object-oriented language makes structural changes a lot easier to handle than conventional imperative programming languages. Every structural item is implemented as a class. When a new part of the structure is needed, such as a connection, it is simply instantiated as an object. The structure of a system is changed by simply adding or removing references to those objects. Of course, this still has to be done in an organised manner.

3.5 Interface

The first interface was implemented as a simple command line program. The user is presented with a list of prompts, asking for various parameters. The program first prompts for information needed to instantiate one of the systems, then for information needed for monitoring and training. The interface is not sufficient to allow detailed testing of the systems. It only prompts for a limited number of parameters, and they cannot be changed once they have been entered.

There was a need for a better interface. The time spent implementing a complex command line interface could be better spent implementing a graphical interface. The Swing components (see section 1.5) made the task of writing a graphical interface relatively easy.

The graphical interface has several advantages over the simple command line program. It is far more flexible, allowing more of the parameters to be changed. The use of the Swing classes makes it easy to add graphical components that represent a parameter, or method. This means that as the systems grow, or different parameters need to be controlled, the interface can easily be extended alongside.

Figure 3.12 shows an example of one of the 3 main system panels. The parameters are entered (although default values are provided), and the “set” method button is pressed, to instantiate the system. This presents buttons for the methods defined on

the class. Currently, the methods available are only those which were necessary for the testing of the systems. New buttons can be added easily in the future.
The method buttons present the user with further options, as shown in figures 3.13, and 3.14.

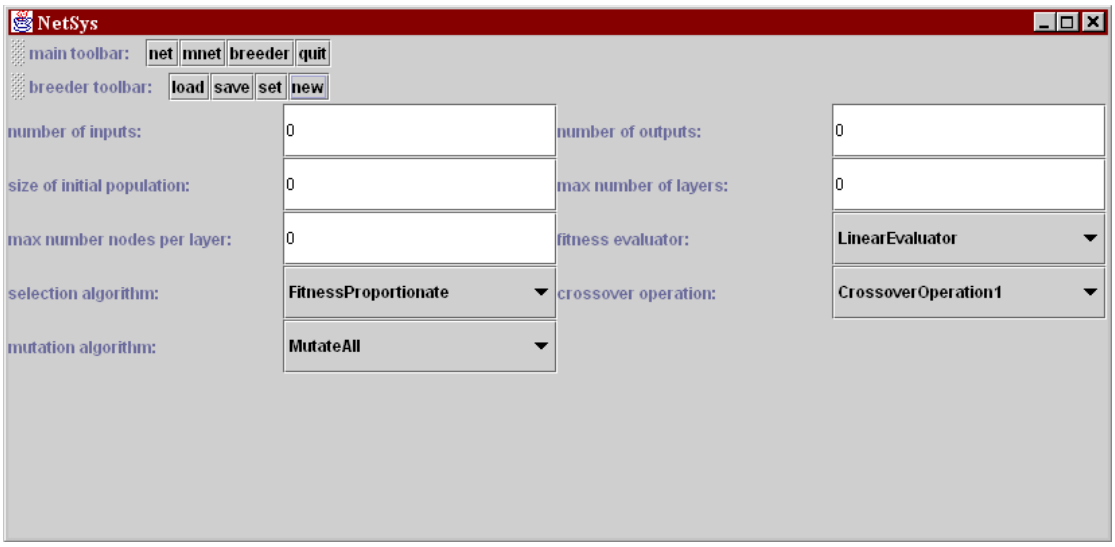


Figure 3.12: The Graphical Interface's FFLNBreeder system panel

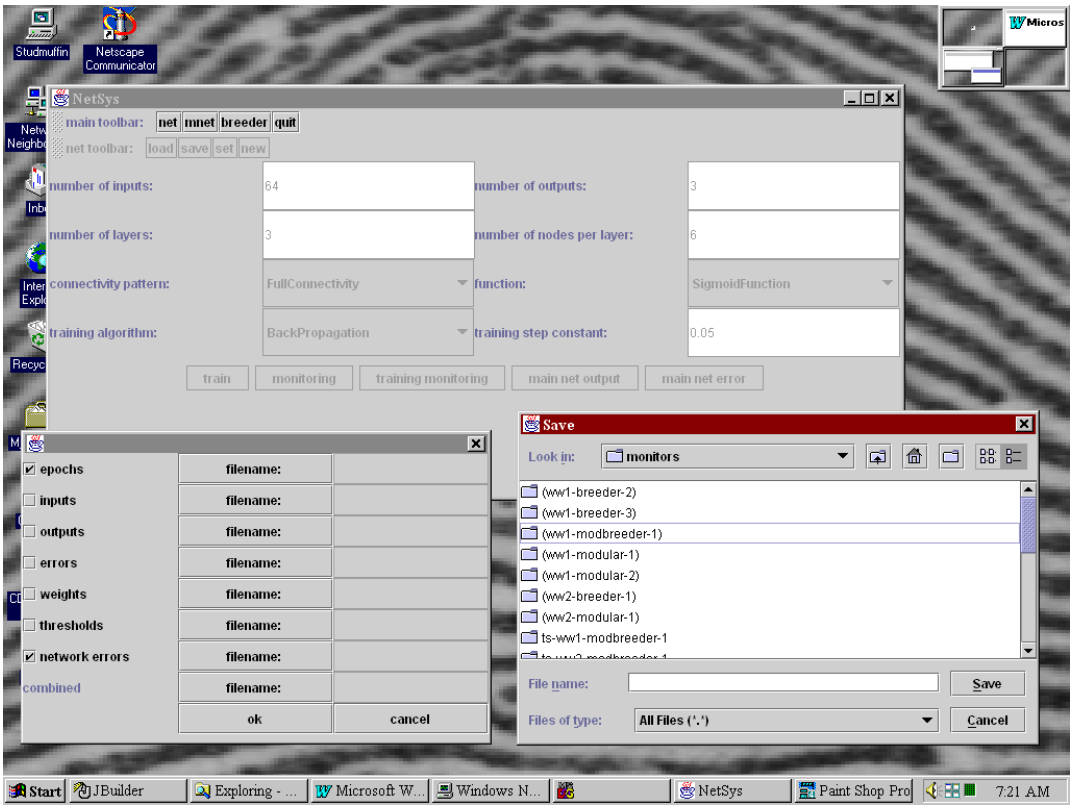


Figure 3.13: The FeedForwardLayeredNetworkMonitor monitor methods panel

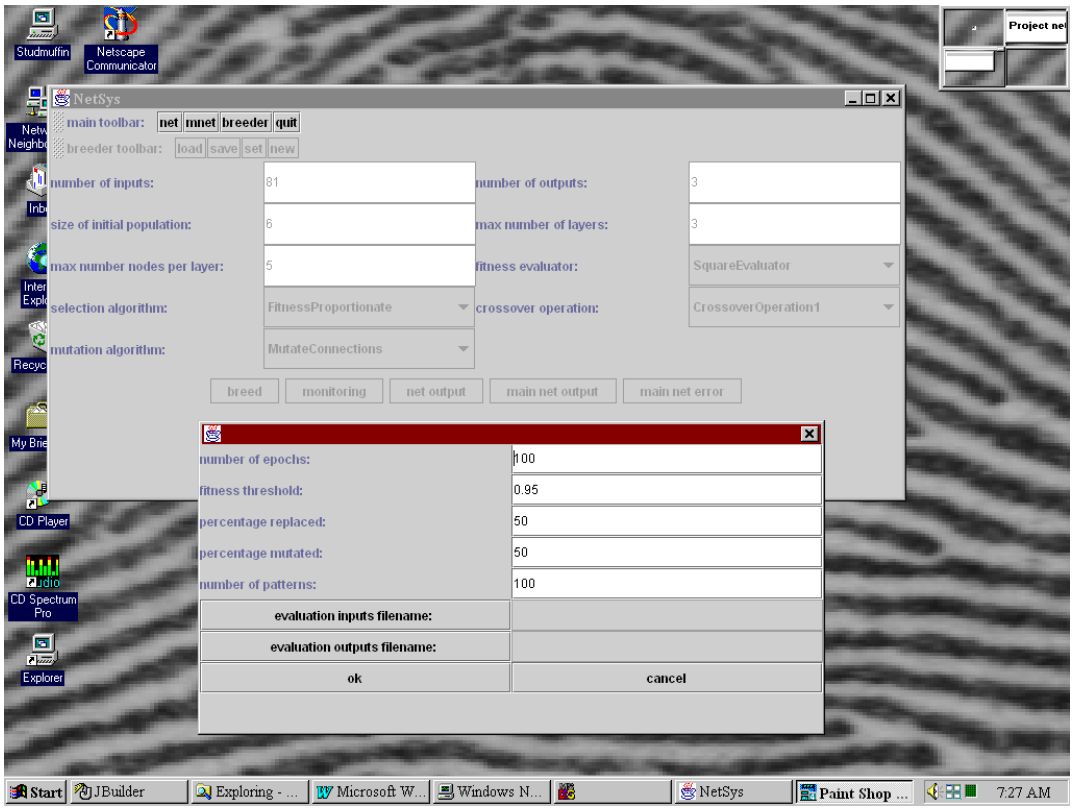


Figure 3.14: The FFLNBreeder train method panel

3.6 Packages

My program code is split into 6 packages. The package name indicates the operation of the classes belonging to it. The packages are as follows,

1 networkPackage: This contains the classes relating to the neural network systems, i.e. FeedForwardLayeredNetwork, ModularNetwork, and any classes they use.
2 breederPackage: This contains all classes related to the breeding systems. These are FFLNBreeder, ModularBreeder, and the classes they use.
3 monitorPackage: This has all of the monitor classes in it.
4 testerPackage: This contains the tester classes.
5 interfacePackage: This contains both the command-line, and graphical interface classes. The command-line interface is implemented using just a single class. Classes implemented for the graphical interface make up the rest of this package.
6 simpleIO: This is a modification of a package written by Chris Kirkham, for use doing simple IO tasks. It is used by the interfaces. I have added a new class called MyText. This is a modification of the existing class Text, and is identical except that rather than catching NumberFormatException exceptions and printing an error message, they are thrown. This is needed so that if a number supplied by the user is of the wrong format, the class using MyText can provide it's own error message.

4. TESTING

The systems must be tested for two reasons:

1. Correctness: To make sure all algorithms are working correctly.
2. Performance: To check the effectiveness of the algorithms.

Correctness tests were carried out throughout the production of the systems. These were necessary to ensure that the classes were working correctly, before being used or extended by other classes. The correctness testing was first carried out using the pre-compiled tests, provided in the testerPackage package. Later, the interfaces could be used to allow flexible testing of the complete systems. All systems have been tested for correctness, and shown to be reliable.

Some performance tests were carried out, giving a first indication as to the success of the systems. However, the running time of the systems is very high. Only a small amount of tests were able to be ran. Further testing is necessary to produce conclusive results.

4.1 Choice of Task

The choice of task is important for both correctness and performance testing. It must allow all aspects of the systems to be tested. In other words, all algorithms must be invoked, and a range of behaviours made possible. This leads to a number of requirements.

1. The task must be sufficiently complex. This is to ensure that the systems have to adapt to perform the task, rather than being initially capable of performing it.

2. The task must have some modularity to it. This means that within the task, there exist sub-tasks, for which if solutions are found, the complete task can be performed.

3. It should be a tried and tested task, for which a capable network structure is known. This is for use verifying that good structures are being produced by the genetic algorithm, and the modular breeder.

4.2 What-Where Vision Task

This task fulfils all of the requirements listed above. It consists of an object-recognition task (the “what” task), and a spatial localisation task (the “where” task). The definition of the what-where task, and the data used to define it, are from a program (and the notes therein) written by my project supervisor, Jonathan Shapiro. It is similar to the task used by Jacobs, Jordan, and Barto (1991) [2], and the similarity should allow suitable expert network structures to be inferred.

4.2.1 The Task

The input is a set of 64 binary numbers. This represents an 8 x 8 matrix. The matrix holds values of 0, except for where a pattern is present. The "what" sub-task asks to distinguish the two following patterns,

1 1 from 1 0
1 1 0 1

The "where" sub-task is to identify which of the four quadrants in the input matrix the pattern is.

There are three output bits. The first gives the solution to the "what" sub-task – 0 indicates the first pattern, and 1 indicates the second pattern. The last two are for the 4 possibilities of the "where" sub-task.

The outputs of the neural networks are real numbers. An output of <0.5 will represents a 0, and an output of ¿ 0.5 will represent a 1.

4.2.2 The Suitable Structures

I believe that the two following networks are the equivalent to the ones presented in [2]. One network is smaller than the other.

The smaller network has a single hidden layer containing 6 nodes, with full-connectivity. The larger network has 12 hidden nodes rather than 6. If the results obtained in [2] provide an accurate indication, this should exhibit a faster learning response.

The modular network will consist of three experts, based on the ones used in [2]. The first two are the networks described above. The third will have no hidden layers, and so all inputs will connect to all outputs.

Network Type	Structure
Neural network 1 Neural network 2	64 → 6 → 3 64 → 12 → 3
Modular network 1 Expert 1 Expert 2 Expert 3	 64 → 12 → 3 64 → 6 → 3 64 → 3

Figure 4.1: The suitable network structures

The network architectures presented above must be tested, and verified to be suitable. Once they have been verified they can be used as a basis for comparison, with both the structures generated by the genetic algorithm, and the modular network breeder.

4.2.2 Crosstalk

A problem affecting network performance when faced with a modular task, is crosstalk [2]. There are two types, temporal and spatial. Temporal crosstalk occurs when a network is trained to perform different tasks, at different times. Spatial crosstalk occurs when a network is trained to perform different tasks at the same time. We are only concerned with spacial crosstalk, as it's a problem present in the what-where task used. The modular network should be more resilient to both types of crosstalk. This is due to its ability to choose which experts are most suitable, and train them in proportion to their error by adjusting their desired output (see section 2.2.2, equation 2.17).

4.3 Results

All systems have been tested on the what-where problem described previously.

1. The neural networks were trained on the task. This serves to verify that the network structures presented in section 4.2.2 can perform adequately.
2. A modular network was similarly trained. This provides a comparison with the performance of a normal neural network.
3. The genetic algorithm was tested. The tests show what changes the populations went through.
4. The modular breeder was tested. This serves to show whether suitable network structures were being produced, and if so, how the system performs.

50 patterns are presented to all systems for training. The number of epochs (or cycles) of a system is plotted against the errors of the system. This gives a good indication of the learning speed. The errors were calculated using different methods, as described in chapter 2, and so the graphs will have different properties. This does not matter, as we are only interested in the number of epochs taken to reach a steady state, not how that state was reached. In order to test a system, it is presented with another 50 different patterns. The percentage of correct answers gives the generalisation ability.

The test parameters are provided in appendix C.

4.3.1 Neural Network Tests

Both neural networks structures (described in section 4.2.2) were tested.

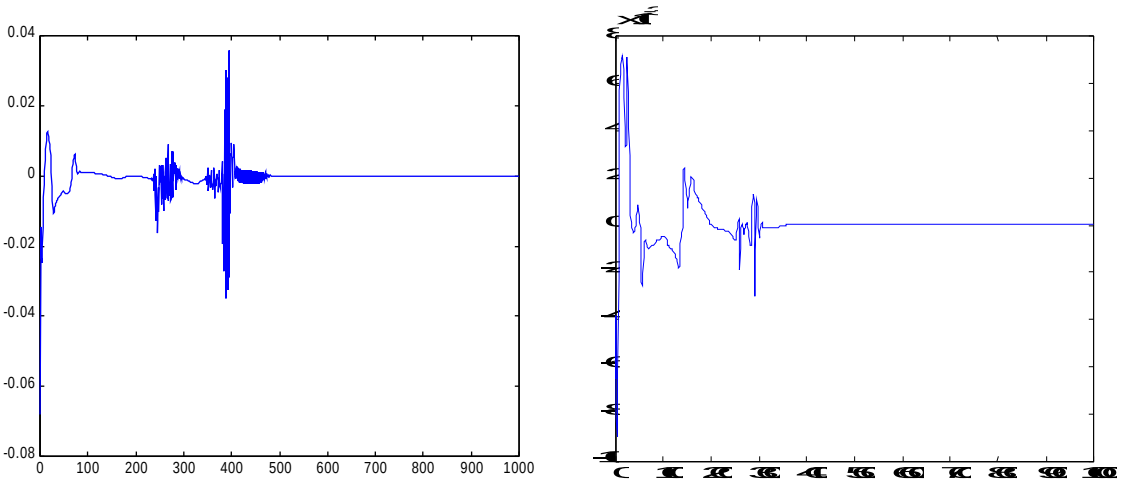
1) Test ww1-network-2: Network 1 is tested.

   The error settled on 0 in 500 epochs. See figure 4.2. The network got 68% of the

testing patterns correct.

2) Test ww1-network-3: Network 2 is tested.

The error reduced to 0 within 400 epochs. See figure 4.2. The network only got 62% of testing patterns correct.



Network 1 Network 2

Figure 4.2: Learning rates of the networks (difference error)

These tests have shown that both networks can learn to perform the task, but display poor ability to generalise. The larger network has a faster learning rate, as expected, but surprisingly is less able to generalise.

4.3.2 Modular Network Test

1) Test ww1-modular-4:

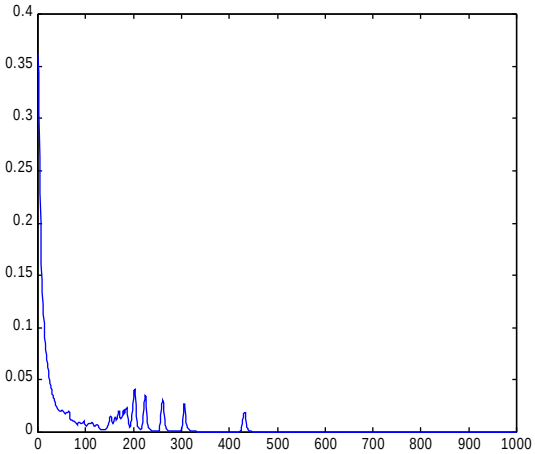


Figure 4.3: Learning rate of the modular network (sum-squared error)

The error settles at ~425 epochs. See figure 4.3. The generalisation ability is considerably better, with 82% of the testing patterns correct.

The system displays a faster learning speed than the smaller of the two neural networks, but not the larger. The good generalisation of the network indicates that the modular network is more robust when faced with crosstalk than the neural networks, as expected.

4.3.2 Genetic Algorithm Tests

Three tests were performed. The second two were performed to try to improve on the results of the first. The random initialisation of networks was prevented from creating any of the competent network structures (as suggested in section 4.2.2). All three of the tests had to be terminated before they had finished. After ~12 hours they were assumed not to be working adequately. As the program was terminated early, the performance on the testing patterns could not be determined.

1) Test ww1-breeder-1:

   Terminated after 30 generations. It uses LinearEvaluator as the fitness evaluator, and MutateAll as the mutation algorithm

   The fitness seemed to be tending toward a value of 0.14. See figure 4.4. However, at around generation 19 it suddenly dropped. This can be explained by looking at the average size of the networks generated, shown in figure 4.5.

   It seems that the average size dropped suddenly on generation 18. The average size in generation 19 is 1, which means there must have been only one population member containing a hidden layer with a node in it. No structures were capable of performing the task, leading to the drop in fitness. The increase in the sizes generated from generation 20 onwards, follows the increase in fitness. The test was terminated before desired fitness could be reached.

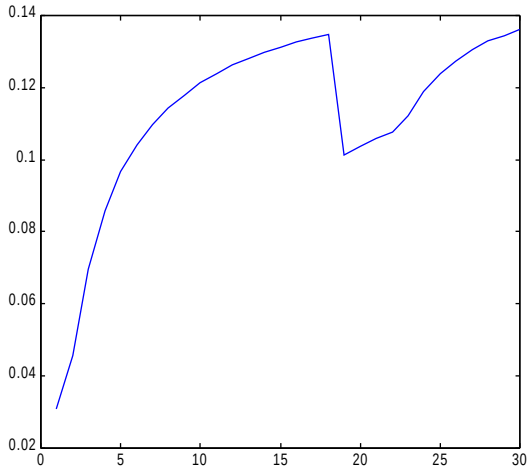


Figure 4.4: Changes in best fitness

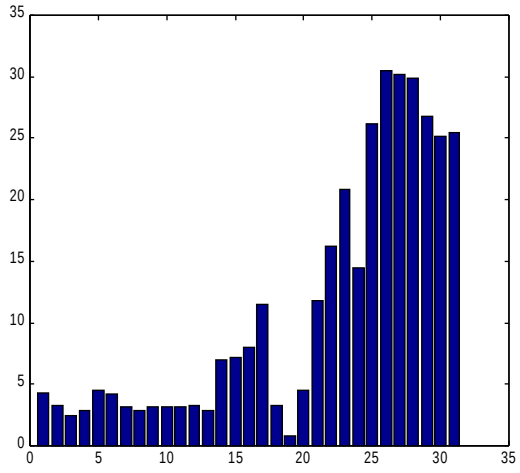


Figure 4.5: Average sizes of networks

2) Test ww1-breeder-4:

Terminated after 30 generations. This changes the fitness evaluator to SquareEvaluator to attempt to select better members. The mutation algorithm has been changed to MutateNodes, to try and prevent such large structural changes. The mutation percentage has been increased to allow a number of (smaller) structural variations. Finally, the fitness threshold has been reduced in an attempt to allow the system to finish naturally.

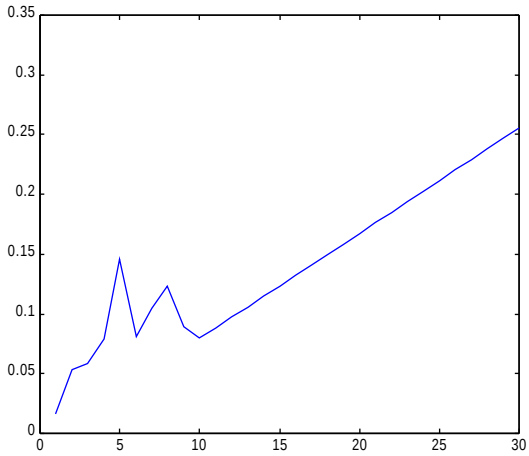


Figure 4.6: Change in best fitness

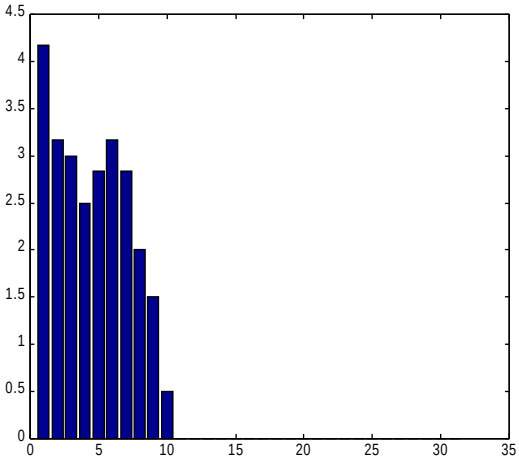


Figure 4.7: Average sizes of networks

The fitness of the best network rises steadily from generation 10 onwards, shown in figure 4.6. A look at the average network sizes (figure 4.7) show that they decreased in size, until from generation 10 onwards there were no networks present containing any hidden nodes.

No suitable network structures were produced by the three genetic algorithm tests. The fitness was shown to rise in both tests. It is possible that suitable structures would have been produced had the systems continued running. However, they involved different changes in network size – test 1 showed the sizes increasing, while test 2 showed them decreasing. The sharper rise of fitness in test 1 compared to the test 2 indicates that network structures averaging a size of 4 are most appropriate. There is a possibility that the network fitness improved due to only the network training.

4.3.3 Modular Breeder

Three tests were run. The output from test 3 is supplied as an example in appendix B.

1) Test ww1-modbreeder-3:

A suitable modular network structure was returned after only two cycles. The patterns were split into 3 sub-sets. The genetic algorithms did not need to make any new populations of experts. This indicates that some of the experts in the

initial population must have already had suitable structures, and the system simply picked them out.

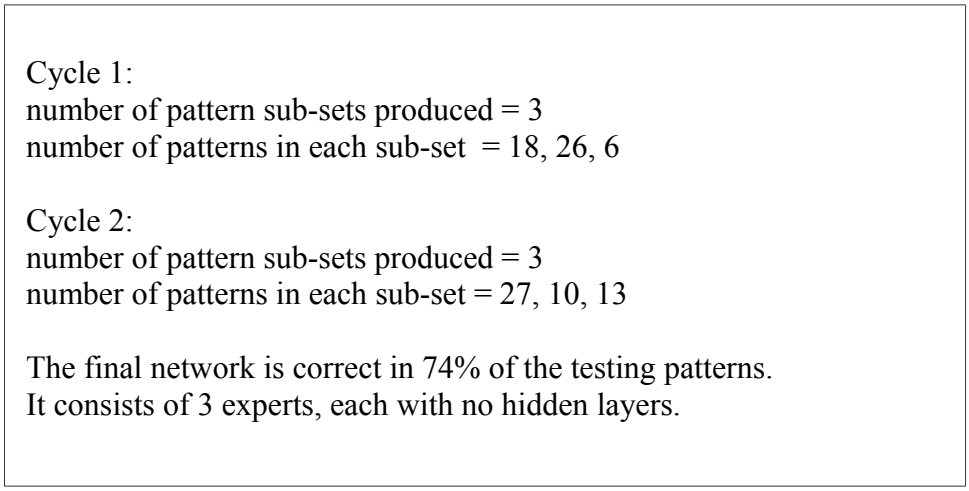


Figure 4.8: Modular breeder results

Each modular network in the system is trained for 100 epochs per cycle. The genetic algorithm trains for 100 epochs per generation. So the system learnt the task on the equivalent of 300 epochs per modular network. This is less than required for the modular network tested above in section 4.3.2, but the generalisation of the network is worse.

2) Test ww1-modbreeder-4:

This test returned a suitable network after only a single cycle. The task was not partitioned. Expert 3 from the first modular network in the population was best on all inputs patterns, and so only a single genetic algorithm was started. Perhaps because of the increased difficulty of performing the whole task, the genetic algorithm took 2 generations until a suitable expert network was returned.

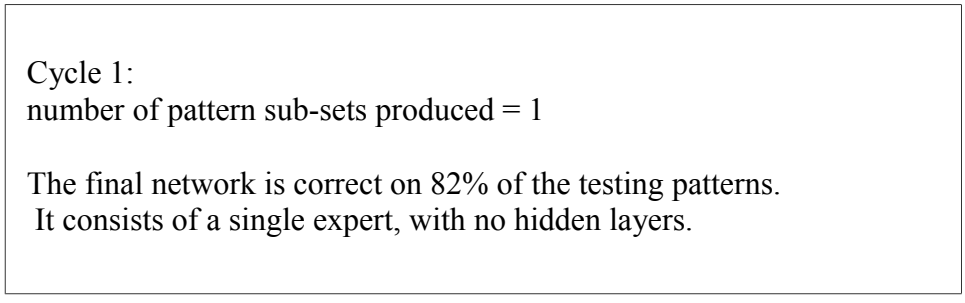


Figure 4.9: Modular breeder results

Each modular network was trained in the equivalent of 300 epochs. The final network learns faster, and can generalise better than the modular network in section 4.3.2.

3) Test ww1-modbreeder-6

This test ran for two cycles. In the first cycle, each of the two genetic algorithms needed two generations to reduce the expert error by the desired amount. In the second cycle, only a single generation was needed, before an expert with the desired error was produced. The task was partitioned into 2 near equal sized sets.

Cycle 1:
number of pattern sub-sets produced = 2
number of patterns in each sub-set = 26, 25

Cycle 2:
number of pattern sub-set produced = 2
number of patterns in each sub-set = 27, 23

The final network is correct on 84% of the testing patterns.
It consists of two experts, each containing no hidden layers.

Figure 4.10: Modular breeder results

The final network got 84% of the testing patterns correct. It consisted of two experts, each containing no hidden layers.

The network was trained in the equivalent of 500 epochs. While this number of epochs is large, the generalisation ability is the best yet.

The modular networks produced by the tests can perform as well as, or better than, the structures presented in section 4.2.2. It's worth noting that the best results were produced by the third test that partitioned the tasks into two near equal sets.

All three of the modular networks returned consist of experts with no hidden layers. This implies,

1. The what-where task specified can be partitioned into sub-tasks that only require encoding of the inputs, and no processing layer.
2. The networks theorised in section 4.2.2 are not good structures. Whilst the tests showed that they can perform the tasks, they are larger than necessary.
3. The second genetic algorithm test may have been producing the best structures (i.e. networks with no hidden layers).

The production of networks containing no hidden nodes is a surprise. Further investigation is required to determine whether it is possible for such structures to perform the task.

5. CONCLUSIONS

This report has followed the development a system to generate the structure of a modular network.

5.1 Achievements

The project has achieved a number of goals. The implementations of the sub-systems are correct, and largely successful. The integration into a complete system has produced good results, although it does not exhibit all desired behaviour.

The classes that define the systems can be reused in other Java programs. They are easy to set up and use. Above all, they are very flexible. Every system has a method to alter nearly every one of its properties. See appendix A for a full listing of the methods.

A framework exists to allow different algorithms to be implemented in the future without needing any alteration to the existing code.

5.1.1 Sub-systems

In order to implement the overall modular breeder system, a number of other systems were developed. These systems make up the bulk of the project. They achieved different degrees of success. They are as follows,

1. The implementation of a neural network and a training algorithm was successful. The training algorithm, Backpropagation, has been shown to be capable of training a neural network. A number of methods have been written that can successfully manipulate the structure of the network.

2. The implementation of a modular network and the main training algorithm, Winner-takes-all, was successful. The algorithm has been shown to produce training results that are superior to that of the neural networks tested.

3. The implementation of a genetic algorithm has been shown to work correctly, although not effectively. The crossover and mutation algorithms provided are the likely cause.

5.1.2 Modular network breeder

The main aim of this project was to produce a system capable of generating the structure of a modular network. Preliminary tests indicate that this aim has been achieved. The modular network breeder returned a suitable network on all of these tests, which exhibited better performance than all the other networks tested. There is no user design of the structure of the network, only decisions as to what size limitations to impose on the initial population.

However, while the system has been shown to produce the correct structure, it does not make effective use of the genetic algorithm. The genetic algorithm simply acts as a further training and selection procedure, without making beneficial structural changes.

The networks structures generated proved very different from the predicted structures. This illustrates a point made in section 1.2 – the structure generated served to give a greater understanding of the problem.

5.1.3 Interface

A simple command-line interface was developed, but proved too limited for the testing of the sub-systems. To provide more control over parameters, a graphical interface was written. This implementation greatly improves the ease at which the systems can be tested. The addition of new method buttons can be done with ease, if there is a desire to add greater control.

5.2 Criticisms

While I wanted experience in writing all of the sub-systems, it would have been beneficial for me to make use of an existing neural network program. I felt it was necessary to implement the neural network to provide the desired structural operations. However, I underestimated the complexity of such a task and in retrospect it would have been better to compromise by using an existing program. This would have allowed me to spend more time on the implementation of the genetic algorithm. A large amount of evolutionary techniques remain unexplored. I believe a more effective system would have been produced if this had been possible.

The performance testing of the systems is incomplete. It serves only as an initial indication of performance. The neural network and modular network were shown to work, but optimal parameters remain to be found. A single run of the genetic algorithm can take over 24 hours. The modular breeder was also slow, due to its use of the genetic algorithm. This meant a great deal of time was needed for correctness testing, leaving less time to test performance.

Fast system running times were not stated as a requirement. However, a faster running time is desirable, if only to permit quicker testing.

5.3 Future Work

5.3.1 Improvements

It would have been useful for the network training algorithms to use a validation set to determine when training should stop. This will lead to better generalisation performance. Also, there is no control of when the training should stop, other than the number of epochs. It would be useful to allow the training to stop when a desired

error has been reached.

The gating network used by the modular network should be allowed to have different inputs than the ones to the modular network itself. This would produce greater flexibility, allowing the gating of the experts to be dependent on different inputs than those defining the task.

The interface is interrupted by any running processes, and is not refreshed until the process has finished. This is inconvenient when the parameters that were set need to be checked. This could be overcome by making every method called from the interface run in its own thread. This would also allow the method to be interrupted, or terminated when desired.

5.3.2 Testing

The correctness testing of the systems served only to verify that their non-public methods function properly when called with expected arguments. This is sufficient to allow the systems to function reliably, as the possible values sent to methods are known. Future testing of these methods is required to ensure that they can cope with unusual values. The public methods, available for use in classes in any packages, have been tested for a greater range of arguments. This is because they are intended for use by people with no knowledge of the implementation details. The non-public methods will only be of use to people who wish to extend the functionality. This is not suggested until a complete technical study has been carried out.

As mentioned previously, performance testing was limited by the running time of the systems. A range of tests need to be performed on the neural network and the modular network, to find optimal parameters – the methods implemented are common, and should work well. More importantly, the genetic algorithm must be further tested in order to determine what algorithms are ineffective. New ones should be written, and different combinations tried. The individual algorithms used by the genetic algorithm are simple, and new ones can be written with ease. It is more the combination of algorithms that present the difficulty. Once more effective neural network generation is possible from the genetic algorithm, the modular network breeder can be properly tested on a harder task than the one previously presented in this report.

5.3.3 Extensions

Methods to load and save a neural network or modular network would be useful. Currently, the graphical interface requires the structure of the experts in a modular network to be specified. Load and save methods would prevent this being necessary. They should be easy to implement, as serialization of objects to disk is part of the functionality of Java.

A large range of different neural network architectures and their associated training algorithms can be implemented. Some examples are Hopfield nets [8 p234], or Recurrent networks [7 p119].

There are many different fitness, selection, crossover, and mutation algorithms for use with the genetic algorithm that can be developed. Some examples that could be useful are niching [11], and fitness sharing [11].

Some performance gathering tools would be useful. This is to allow the performance of the networks to be easily retrieved. Currently, the files generated by the monitors are read into MATLAB as matrices. Work has to be done editing the matrices in order to display certain results. It would be useful to implement a set of methods that automatically format the monitor data. These can be implemented by the monitor classes themselves.

REFERENCES

[1] Haykin, S. (1999). Neural Networks: A Comprehensive Foundation (2nd edition). pp368. *Prentice Hall*.

[2] Jacobs, R.A., Jordan, M.I., & Barto, A.G. (1991). Task Decomposition Through Competition in a Modular Connectionist Architecture: The What and Where Vision Tasks. *Cognitive Science, 15, 219-250*.

[3] Jacobs, R., Jordan, M., Nowlan, S., & Hinton, G. (1991). Adaptive Mixtures of Local Experts. *Neural Computation, 3, 79-87*.

[4] Jacobs, R., & Jordan, M.. (1993). Hierarchical Mixture of Experts & the EM Algorithm. *Neural Compuation, 6 (2), 181-214.*

[5] Liu, Y.,& Yao, X. (?). Evolving Neural Networks Which Generalise Well. *Computational Intelligence Group, School of Computer Science, University College, The University of New South Wales. Australian Defence Force Acadamy, Canberra, ACT, Australia 2600*.

[6] MacLeod, C.,& Maxwell, G. (1999). Growing Artificial Neural Networks using Evolutionary Algorithms. *School of Electronic & Electrical Engineering, The Robert Gordon Luniversity, Aberdeen, Scotland*.

[7] Mitchell, T.M. (1997). Machine Learning. pp250. *McGraw-Hill*.

[8] Pratt, I. (1994). Artificial Intelligence. pp224. *The Macmillan Press*.

[9] Ramamurti, V.,& Ghosh, J. (?). On the Use of Localized Gating in Mixture of Experts Networks. *SBC Technology Resources, Inc., 9505 Arboretum Blvd., Austin TX 78759, USA. Department of Electrical and Computer Engineering, UT Austin, Austin TX 78712, USA*.

[10] Ramamurti, V.,& Ghosh, J. (?). Structurally Adaptive Localized Mixtures of Experts for Non-Stationary Environments. *Department of Electrical and Computer Engineering. UT Austin. TX 78712-1084, USA*.

[11] Yao, X.,& Darwen, P. (1996). Speciation as Automatic Categorial Modulariztion. *IEEE Transactions on Evolutionary Compuation*.

APPENDIX A – Public class and method listing

This appendix lists all public classes and methods needed to make full use of the algorithms provided. It is by not a full listing of all classes or the methods they provide.

1. networkPackage

```
public class AllButOne extends ConnectivityPattern implements Serializable
        public AllButOne(int _numNodeToBeLeftOut)
        public ConnectivityPattern copy()

public class BackPropagation extends FFLNTrainingAlgorithm implements Serializable
        public BackPropagation(double _constant, DoubleFunction _function)
        public BackPropagation(DoubleFunction _function)
        public BackPropagation(double _constant)
        public BackPropagation()
        public void setConstant(double _constant)
        public double getConstant()
        public void setFunction(DoubleFunction _function)
        public DoubleFunction getFunction()
        public void trainNetwork(FeedForwardLayeredNetwork _network, double [][] _inputs, double
                [][] _desiredOutputs, int _numEpochs)
                public void setMonitor(BackPropagationMonitor _monitor)
        public BackPropagationMonitor getMonitor()
        public FFLNTrainingAlgorithm copy()

public class DifferentiatedSigmoid extends DoubleFunction implements Serializable
        public DifferentiatedSigmoid()
        public DifferentiatedSigmoid(double _scalingConstant)
        public DoubleFunction copy()

public class DivideFunction extends Function implements Serializable
        public Function copy()

public class ErrorAdjust extends MNTrainingAlgorithm implements Serializable
        public ErrorAdjust(double _constant)
        public ErrorAdjust()
        public void setConstant(double _constant)
        public void trainNetwork(ModularNetwork _network, double [][] _inputs, double [][]
                _desiredOutputs, int _numEpochs)
        public MNTrainingAlgorithm copy()
        public void setMonitor(ErrorAdjustMonitor _monitor)
        public ErrorAdjustMonitor getMonitor()

public class ExponentialFunction extends Function implements Serializable
        public Function copy()

public class FeedForwardLayeredNetwork extends Network implements Serializable
        public FeedForwardLayeredNetwork(int _numInputs, int [] _numNodesPerHiddenLayer,
                Function [][] _nodesFunction, ConnectivityPattern [][] _nodesConnectivityPattern,
                int _numOutputs, Function [] _outputsFunction, ConnectivityPattern []
                _outputsConnectivityPattern, FFLNTrainingAlgorithm _trainingAlgorithm)
        public FeedForwardLayeredNetwork(int _numInputs, int [] _numNodesPerHiddenLayer,
                Function [] _nodesFunction, ConnectivityPattern [] _nodesConnectivityPattern, int
                _numOutputs, Function _outputsFunction, ConnectivityPattern
                _outputsConnectivityPattern, FFLNTrainingAlgorithm _trainingAlgorithm)
```

```
public FeedForwardLayeredNetwork(int _numInputs, int [] _numNodesPerHiddenLayer, int
        _numOutputs, Function _function, ConnectivityPattern _connectivityPattern,
        FFLNTrainingAlgorithm _trainingAlgorithm)
public FeedForwardLayeredNetwork(int _numInputs, int [] _numNodesPerHiddenLayer, int
        _numOutputs, ConnectivityPattern _connectivityPattern)
public FeedForwardLayeredNetwork(int _numInputs, int _numOutputs)
public int getNumberOfInputs()
public int getNumberOfOutputs()
public int getNumberOfLayers()
public int getNumberOfNodes()
public int getNumberOfActiveNodes()
public int getNumberOfNodesInLayer(int _layerNumber)
public int getNumberOfActiveNodesInLayer(int _layerNumber)
public double [] getNetworkOutput(double [] inputValues)
public double [] getOutputActivations()
public double getOutputActivation(int _outputNumber)
public double getNodeActivation(int _layerNumber, int _nodeNumber)
public void insertLayerAt(int _insertPosition)
public void addLayer()
public void removeLayer(int _layerNumber)
public int [] getInputConnectionsToLayerAbove(int _inputNumber)
public boolean nodeActive(int _layerNumber, int _nodeNumber)
public void activateNode(int _layerNumber, int _nodeNumber)
public void deactivateNode(int _layerNumber, int _nodeNumber)
public void addNode(int layerNumber, Function function, ConnectivityPattern
        _connectivityPattern)
public void addNode(int layerNumber, ConnectivityPattern _connectivityPattern)
public void addNode(int layerNumber, Function function)
public void addNode(int layerNumber)
public void addNode(Function function, ConnectivityPattern _connectivityPattern)
public void addNode(Function function)
public void removeNode(int _layerNumber, int _nodeNumber)
public Function copyHiddenNodeFunction(int _layerNumber, int _nodeNumber)
public int [] getHiddenNodeConnectionsToLayerAbove(int _layerNumber, int _nodeNumber)
public ConnectivityPattern copyHiddenNodeConnectivityPattern(int _layerNumber, int
        _nodeNumber)
void setHiddenNodeConnectivityPattern(int _layerNumber, int _nodeNumber,
        ConnectivityPattern _pattern)
public int getNumberOfHiddenNodeConnections(int _layerNumber, int _nodeNumber)
public int [] getHiddenNodeConnections(int _layerNumber, int _nodeNumber)
public void setHiddenNodeConnections(int _layerNumber, int _nodeNumber, int []
        connections)
public int getHiddenNodeConnection(int _layerNumber, int _nodeNumber, int
        _connectionNumber)
public void addConnectionToHiddenNode(int _layerNumber, int _nodeNumber, int
        connection)
public boolean removeConnectionFromHiddenNode(int _layerNumber, int _nodeNumber, int
        _connectionNumber)
public double [] getHiddenNodeWeights(int _layerNumber, int _nodeNumber)
public boolean setHiddenNodeWeights(int _layerNumber, int _nodeNumber, double []
        weights)
public double getHiddenNodeWeight(int _layerNumber, int _nodeNumber, int
        _connectionNumber)
public boolean setHiddenNodeWeight(int _layerNumber, int _nodeNumber, int
        _connectionNumber, double _weight)
public double getHiddenNodeThreshold(int _layerNumber, int _nodeNumber)
public void setHiddenNodeThreshold(int _layerNumber, int _nodeNumber, double
        _threshold)
public Function copyOutputFunction(int _outputNumber)
public ConnectivityPattern copyOutputConnectivityPattern(int _outputNumber)
```

```
        public void setOutputConnectivityPattern(int _outputNumber, ConnectivityPattern _pattern)
        public int getNumberOfOutputConnections(int _outputNumber)
        public int [] getOutputConnections(int _outputNumber)
        public void setOutputConnections(int _outputNumber, int [] _connections)
        public int getOutputConnection(int _outputNumber, int _connectionNumber)
        public void addConnectionToOutputNode(int _outputNumber, int _connection)
        public boolean removeConnectionFromOutputNode(int _outputNumber, int
                _connectionNumber)
        public double [] getOutputWeights(int _outputNumber)
        public boolean setOutputWeights(int _outputNumber, double [] _weights)
        public double getOutputWeight(int _outputNumber, int _connectionNumber)
        public double getOutputThreshold(int _outputNumber)
        public void setOutputThreshold(int _outputNumber, double _threshold)
        public void train(double [][] _inputs, double [][] _desiredOutputs, int _numEpochs)
        public void setTrainingAlgorithm(FFLNTrainingAlgorithm algorithm)
        public FFLNTrainingAlgorithm copyTrainingAlgorithm()
        public void setMonitor(FeedForwardLayeredNetworkMonitor _monitor)
        public FeedForwardLayeredNetworkMonitor getMonitor()

public class FullConnectivity extends ConnectivityPattern implements Serializable
        public ConnectivityPattern copy()

public class ModularNetwork extends Network implements Serializable
        public ModularNetwork(int _numInputs, int _numOutputs, Network [] _networks,
                MNTrainingAlgorithm _trainingAlgorithm)
        public ModularNetwork(int _numInputs, int _numOutputs, Network [] _networks)
        public double [] getNetworkOutput(double [] _inputs)
        public int getNumberOfInputs()
        public int getNumberOfOutputs()
        public int getNumberOfNetworkModules()
        public Network getNetworkModule(int _networkNumber)
        public Network [] getNetworkModules()
        public void addNetworkModule(Network network)
        public void removeNetworkModule(int _networkNumber)
        public double [] getWeightsOfSoftmaxNode(int _nodeNumber)
        public void setWeightsOfSoftmaxNode(int _nodeNumber, double [] _weights)
        public double [] getModuleWeights(double [] _inputs)
        public double [] getModuleWeights()
        public double getModuleWeight(int _numModule, double [] _inputs)
        public double getModuleWeight(int _numModule)
        public void setTrainingAlgorithm(MNTrainingAlgorithm algorithm)
        public MNTrainingAlgorithm getTrainingAlgorithm()
        public void train(double [][] _inputs, double [][] _desiredOutputs, int _numEpochs)
        public void setMonitor(ModularNetworkMonitor _monitor)
        public ModularNetworkMonitor getMonitor()

public class NullFunction extends Function implements Serializable
        public Function copy()

public class OnlyOne extends ConnectivityPattern implements Serializable
        public OnlyOne(int _numOfOnlyNode)
        public ConnectivityPattern copy()

public class OnlySome extends ConnectivityPattern implements Serializable
        public OnlySome(int [] theOnes)
        public ConnectivityPattern copy()

public class PatternFileReader
        public double [][] convertFile(int _numPatterns, int _numElements, String _filename) throws
                FileNotFoundException, IOException
```

public double [][] convertFile(int _numPatterns, int _numElements, File _file) throws FileNotFoundException, IOException

public class SigmoidFunction extends Function implements Serializable
public SigmoidFunction(double _scalingConstant)
public SigmoidFunction()
public Function copy()

public class SoftMaxFunction extends Function implements Serializable
public Function copy()

public class SumFunction extends Function implements Serializable
public Function copy()

public class UserDefinedConnectivity extends ConnectivityPattern implements Serializable
public UserDefinedConnectivity()
public ConnectivityPattern copy()

public class WinnerTakesAll extends MNTrainingAlgorithm implements Serializable
public WinnerTakesAll(double _memoryConstant, double _improvConstant, double _adjustConstant)
public WinnerTakesAll()
public void setConstants(double _memoryConstant, double _improvConstant, double _adjustConstant)
public void trainNetwork(ModularNetwork _network, double [][] _inputs, double [][] _desiredOutputs, int _numEpochs)
public MNTrainingAlgorithm copy()
public void setMonitor(WinnerTakesAllMonitor _monitor)
public WinnerTakesAllMonitor getMonitor()

2. breederPackage

public class CrossoverOperation1 extends CrossoverOperation
public void setMonitor(CrossoverOperation1Monitor _monitor)
public CrossoverOperation1Monitor getMonitor()

public class FFLNBreeder
public FFLNBreeder(FeedForwardLayeredNetwork [] _initialPopulation, CrossoverOperation _crossoverOp, FitnessEvaluator _fitnessOp, SelectionAlgorithm _selectionOp, MutationAlgorithm _mutationOp)
public FFLNBreeder(int _numInputs, int _numOutputs, int _size, int _maxNumLayers, int _maxNumNodesPerLayer, CrossoverOperation _crossoverOp, FitnessEvaluator _fitnessOp, SelectionAlgorithm _selectionOp, MutationAlgorithm _mutationOp)
public int getNumberOfInputs()
public int getNumberOfOutputs()
public int getPopulationSize()
public FeedForwardLayeredNetwork breed(int _numEpochs, double _fitnessThreshold, int _percentageReplaced, int _mutationRate, double [][] _evaluationInputs, double [][] _evaluationOutputs)
public FeedForwardLayeredNetwork [] getPopulation()
public FeedForwardLayeredNetwork getMember(int _networkNumber)
public FeedForwardLayeredNetwork getBestNetwork()
public int bestMember()
public double getMembersFitness(int _networkNumber)
public double [] getMembersFitnesses()
public MutationAlgorithm getMutationAlgorithm()
public void setMutationAlgorithm(MutationAlgorithm _op)
public CrossoverOperation getCrossoverOperation()
public void setCrossoverOperation(CrossoverOperation _op)

```
        public FitnessEvaluator getFitnessEvaluator()
        public void setFitnessEvaluator(FitnessEvaluator _evaluator)
        public SelectionAlgorithm getSelectionAlgorithm()
        public void setSelectionAlgorithm(SelectionAlgorithm _algorithm)
        public void setMonitor(FFLNBreederMonitor _monitor)
        public FFLNBreederMonitor getMonitor()

public class FitnessProportionate extends SelectionAlgorithm
        public void setMonitor(FitnessProportionateMonitor _monitor)
        public FitnessProportionateMonitor getMonitor()

public class InverseErrorEvaluator extends FitnessEvaluator
        public InverseErrorEvaluator(double _power)
        public void setMonitor(InverseErrorEvaluatorMonitor _monitor)
        InverseErrorEvaluatorMonitor getMonitor()

public class LinearEvaluator extends InverseErrorEvaluator
        public LinearEvaluator()

public class ModularBreeder
        public ModularBreeder(int _numInputs, int _numOutputs, int _numModNets, int
                _initialNumModules, int _maxNumLayers, int _maxNumNodesPerLayer)
        public ModularNetwork breedModularNetwork(double [][] _inputs, double [][]
                _desiredOutputs, int _numModularEpochs, int _numBreederEpochs, double
                _desiredError, int _percentageImprovement,  double _backpropStepConstant, double
                _memoryConstant, double _stepConstant, double _improvementConstant, int
                _percentageReplaced, int _percentageMutated)
        public void setMonitor(ModularBreederMonitor _monitor)
        public ModularBreederMonitor getMonitor()

public class MutateAll extends MutationAlgorithm
        public void setMonitor(MutateAllMonitor _monitor)
        public MutateAllMonitor getMonitor()

public class MutateAllButConnections extends MutationAlgorithm
        public void setMonitor(MutateAllMonitor _monitor)
        public MutateAllMonitor getMonitor()

public class MutateConnections extends MutationAlgorithm
        public void setMonitor(MutateConnectionsMonitor _monitor)
        public MutateConnectionsMonitor getMonitor()
public class MutateNodes extends MutationAlgorithm
        public void setMonitor(MutateNodesMonitor _monitor)
        public MutateNodesMonitor getMonitor()

public class NodeDeactivationMutation extends MutationAlgorithm
        public void setMonitor(NodeDeactivationMutationMonitor _monitor)
        public NodeDeactivationMutationMonitor getMonitor()
```

3. monitorPackage

```
public class BackPropagationMonitor extends ActivityMonitor
        public BackPropagationMonitor(String _pathname) throws IOException
        public BackPropagationMonitor()
        public boolean monitorAll(boolean bool)
        public boolean monitorEpochs(String _pathname) throws IOException
        public boolean monitorEpochs(boolean bool)
        public boolean monitorInputs(String _pathname) throws IOException
```

public boolean monitorInputs(boolean bool)
public boolean monitorOutputs(String _pathname) throws IOException
public boolean monitorOutputs(boolean bool)
public boolean monitorErrors(String _pathname) throws IOException
public boolean monitorErrors(boolean bool)
public boolean monitorWeights(String _pathname) throws IOException
public boolean monitorWeights(boolean bool)
public boolean monitorThresholds(String _pathname) throws IOException
public boolean monitorThresholds(boolean bool)
public boolean monitorNetworkErrors(String _pathname) throws IOException
public boolean monitorNetworkErrors(boolean bool)

public class CrossoverOperation1Monitor extends ActivityMonitor
public CrossoverOperation1Monitor(String _pathname) throws IOException
public CrossoverOperation1Monitor()
public boolean monitorAll(boolean bool)
public boolean monitorLayers(String _pathname) throws IOException
public boolean monitorLayers(boolean bool)
public boolean monitorCuts(String _pathname) throws IOException
public boolean monitorCuts(boolean bool)
public boolean monitorChildren(String _pathname) throws IOException
public boolean monitorChildren(boolean bool)

public class ErrorAdjustMonitor extends ModularTrainingMonitor
public ErrorAdjustMonitor(String _pathname) throws IOException
public ErrorAdjustMonitor()

public class FeedForwardLayeredNetworkMonitor extends ActivityMonitor
public FeedForwardLayeredNetworkMonitor(String _pathname) throws IOException
public FeedForwardLayeredNetworkMonitor()
public boolean monitorAll(boolean bool)
public boolean monitorLayers(String _pathname) throws IOException
public boolean monitorLayers(boolean bool)
public boolean monitorNodes(String _pathname) throws IOException
public boolean monitorNodes(boolean bool)
public boolean monitorWeights(String _pathname) throws IOException
public boolean monitorWeights(boolean bool)
public boolean monitorThresholds(String _pathname) throws IOException
public boolean monitorThresholds(boolean bool)

public class FFLNBreederMonitor extends ActivityMonitor
public FFLNBreederMonitor(String _pathname) throws IOException
public FFLNBreederMonitor()
public boolean monitorAll(boolean bool)
public boolean monitorGenerations(String _pathname) throws IOException
public boolean monitorGenerations(boolean bool)
public boolean monitorBest(String _pathname) throws IOException
public boolean monitorBest(boolean bool)
public boolean monitorFitness(String _pathname) throws IOException
public boolean monitorFitness(boolean bool)
public boolean monitorPopulation(String _pathname) throws IOException
public boolean monitorPopulation(boolean bool)
public boolean monitorNetworks(String _pathname) throws IOException
public boolean monitorNetworks(boolean bool)

public class FitnessProportionateMonitor extends ActivityMonitor
public FitnessProportionateMonitor(String _pathname) throws IOException
public FitnessProportionateMonitor()
public boolean monitorAll(boolean bool)

public class InverseErrorEvaluatorMonitor extends ActivityMonitor
    public InverseErrorEvaluatorMonitor(String _pathname) throws IOException
    public InverseErrorEvaluatorMonitor()
    public boolean monitorAll(boolean bool)

public class ModularBreederMonitor extends ActivityMonitor
    public ModularBreederMonitor(String _pathname) throws IOException
    public ModularBreederMonitor()
    public boolean monitorAll(boolean bool)
    public boolean monitorCycles(String _pathname) throws IOException
    public boolean monitorCycles(boolean bool)
    public boolean monitorErrors(String _pathname) throws IOException
    public boolean monitorErrors(boolean bool)

public class ModularNetworkMonitor extends ActivityMonitor
    public ModularNetworkMonitor(String _pathname) throws IOException
    public ModularNetworkMonitor()
    public boolean monitorAll(boolean bool)
    public boolean monitorModules(String _pathname) throws IOException
    public boolean monitorModules(boolean bool)
    public boolean monitorModuleWeights(String _pathname) throws IOException
    public boolean monitorModuleWeights(boolean bool)
    public boolean monitorWeights(String _pathname) throws IOException
    public boolean monitorWeights(boolean bool)

public class ModularTrainingMonitor extends ActivityMonitor
    public ModularTrainingMonitor(String _pathname) throws IOException
    public boolean monitorAll(boolean bool)
    public boolean monitorEpochs(String _pathname) throws IOException
    public boolean monitorEpochs(boolean bool)
    public boolean monitorInputs(String _pathname) throws IOException
    public boolean monitorInputs(boolean bool)
    public boolean monitorErrors(String _pathname) throws IOException
    public boolean monitorErrors(boolean bool)
    public boolean monitorSoftmaxWeights(String _pathname) throws IOException
    public boolean monitorSoftmaxWeights(boolean bool)
    public boolean monitorWinningNetworks(String _pathname) throws IOException
    public boolean monitorWinningNetworks(boolean bool)
    public boolean monitorAverageErrors(String _pathname) throws IOException
    public boolean monitorAverageErrors(boolean bool)
    public boolean monitorAverageWeights(String _pathname) throws IOException
    public boolean monitorAverageWeights(boolean bool)

public class MutateAllMonitor extends ActivityMonitor
    public MutateAllMonitor(String _pathname) throws IOException
    public MutateAllMonitor()
    public boolean monitorAll(boolean bool)
    public boolean monitorLayers(String _pathname) throws IOException
    public boolean monitorLayers(boolean bool)
    public boolean monitorNodes(String _pathname) throws IOException
    public boolean monitorNodes(boolean bool)
    public boolean monitorConnections(String _pathname) throws IOException
    public boolean monitorConnections(boolean bool)

public class MutateConnectionsMonitor extends ActivityMonitor
    public MutateConnectionsMonitor(String _pathname) throws IOException
    public MutateConnectionsMonitor()
    public boolean monitorAll(boolean bool)

public class MutateNodesMonitor extends ActivityMonitor

public MutateNodesMonitor(String _pathname) throws IOException
public MutateNodesMonitor()
public boolean monitorAll(boolean bool)

public class NodeDeactivationMutationMonitor extends ActivityMonitor
public NodeDeactivationMutationMonitor(String _pathname) throws IOException
public NodeDeactivationMutationMonitor()
public boolean monitorAll(boolean bool)

public class WinnerTakesAllMonitor extends ModularTrainingMonitor
public WinnerTakesAllMonitor(String _pathname) throws IOException
public WinnerTakesAllMonitor()
public boolean monitorAll(boolean bool)
public boolean monitorImprovement(String _pathname) throws IOException
public boolean monitorImprovement(boolean bool)

4. interfacePackage

public class Interface
public static void main(String [] args)

public class NetSys
public static void main(String [] args)

APPENDIX B – Sample code output

```
SAMPLE OUTPUT FROM TEST WW1-MODBREEDER-6

AppAccelerator(tm) 1.1.034 for Java (JDK
1.1), x86 version.
Copyright (c) 1998 Borland International.
All Rights Reserved.

Welcome to the network breeder (type
'help' for a list of commands).
-> modbreeder

modbreeder - set up a modular network
breeder, and breed modular networks.

please enter following parameters (space
seperated);
number of inputs: 64
number of outputs: 3
number of modular networks: 5
initial number of modules in each network:
3
maximun number of layers in randomly
generated networks: 2
maximum number of nodes per layer in
randomly generated networks: 5
number of training patterns: 50
filename of training inputs data:
h:\uni\project\data\what-where\ww1-inputs
filename of training outputs data:
h:\uni\project\data\what-where\ww1-outputs
number of testing patterns: 100
filename of testing inputs data:
h:\uni\project\data\what-where\ww1-test-
inputs
filename of testing outputs data:
h:\uni\project\data\what-where\ww1-test-
outputs
number of epochs to train modular
networks: 50
number of epochs to train each population
member: 50
desired error: 0.1
percentage improvement that breeder must
achieve: 30
module training step constant: 0.1
modular network training memory constant:
0.1
modular network training step constant:
0.1
modular network training improvement
constant: 1.0
percentage of population to replace: 50
percentage of population to mutate: 50
monitoring on? (y/n): y
filename for cycle monitoring:
h:\uni\project\monitor\ww1-modbreeder-
6\ww1-modbreeder-cycles-6
IO exception, probably incorrect filename
filename for cycle monitoring:
h:\uni\project\monitors\ww1-modbreeder-
6\ww1-modbreeder-cycles-6
filename for error monitoring:
h:\uni\project\monitors\ww1-modbreeder-
6\ww1-modbreeder-errors-6
evolving modular networks to create best
network...

setting up initial population of modular
networks


training all modular networks

TEST: training all modules and gating
network, on all examples (for 1 epoch per
example), for 50 epochs

m.....................................
.....................................
-EDITED-
.............................
TEST: average error per epoch of module 1
= 0.41409505383830586
TEST: average weight per epoch of module
1 = 0.05723214476484701
TEST: average error per epoch of module 2
= 0.1977500647031548
TEST: average weight per epoch of module
2 = 0.48765896674784465
TEST: average error per epoch of module 3
= 0.2283812008382145
TEST: average weight per epoch of module
3 = 0.45538084793551853

TEST: training all modules and gating
network, on all examples (for 1 epoch per
example), for 50 epochs

m.....................................
.....................................
-EDITED-
.............................
TEST: average error per epoch of module 1
= 0.3593454122047755
TEST: average weight per epoch of module
1 = 0.21801218370762906
TEST: average error per epoch of module 2
= 0.3948502128677635
TEST: average weight per epoch of module
2 = 0.21695961178077497
TEST: average error per epoch of module 3
= 0.2591488590363779
TEST: average weight per epoch of module
3 = 0.27257606758522274

TEST: training all modules and gating
network, on all examples (for 1 epoch per
example), for 50 epochs

m.....................................
.....................................
-EDITED-
.............................
TEST: average error per epoch of module 1
= 0.29996790608531915
TEST: average weight per epoch of module
1 = 0.21297670070064267
TEST: average error per epoch of module 2
= 0.3322266611440392
TEST: average weight per epoch of module
2 = 0.2014497783024335
TEST: average error per epoch of module 3
= 0.33674762989583235
TEST: average weight per epoch of module
3 = 0.26253665816305155

TEST: training all modules and gating
network, on all examples (for 1 epoch per
example), for 50 epochs

m.....................................
.....................................
-EDITED-
.............................
```

```
TEST: average error per epoch of module 1
= 0.4052036345450615
TEST: average weight per epoch of module 1
= 0.12937120456648465
TEST: average error per epoch of module 2
= 0.4736577133796439
TEST: average weight per epoch of module 2
= 0.1417863601427705
TEST: average error per epoch of module 3
= 0.07887762463444313
TEST: average weight per epoch of module 3
= 0.7285799989086873

TEST: training all modules and gating
network, on all examples (for 1 epoch per
example), for 50 epochs

m.....................................
.....................................
-EDITED-
.............................
TEST: average error per epoch of module 1
= 0.3767399831623017
TEST: average weight per epoch of module 1
= 0.2147852284321644
TEST: average error per epoch of module 2
= 0.36932931980672207
TEST: average weight per epoch of module 2
= 0.23905104781952322
TEST: average error per epoch of module 3
= 0.42020080128007664
TEST: average weight per epoch of module 3
= 0.20947016996358023

setting best modules for each input
pattern

input set 1 best modules are: 2 3 1 3 1
input set 2 best modules are: 2 3 2 3 1
input set 3 best modules are: 2 3 3 3 3
input set 4 best modules are: 2 3 3 3 1
input set 5 best modules are: 2 3 1 3 3
input set 6 best modules are: 2 3 3 3 1
input set 7 best modules are: 3 3 2 3 2
input set 8 best modules are: 3 3 2 3 2
input set 9 best modules are: 3 3 2 2 2
input set 10 best modules are: 3 3 2 3 2
input set 11 best modules are: 2 3 3 1 1
input set 12 best modules are: 2 3 3 3 1
input set 13 best modules are: 2 3 1 1 1
input set 14 best modules are: 2 3 1 3 1
input set 15 best modules are: 3 3 1 2 2
input set 16 best modules are: 3 3 2 3 2
input set 17 best modules are: 3 3 2 3 2
input set 18 best modules are: 3 3 2 3 2
input set 19 best modules are: 2 3 3 1 1
input set 20 best modules are: 2 3 3 3 1
input set 21 best modules are: 2 3 3 1 1
input set 22 best modules are: 2 3 3 3 1
input set 23 best modules are: 3 3 2 2 2
input set 24 best modules are: 3 3 2 3 2
input set 25 best modules are: 3 3 2 2 2
input set 26 best modules are: 3 3 2 3 2
input set 27 best modules are: 2 3 1 1 1
input set 28 best modules are: 2 3 1 3 1
input set 29 best modules are: 2 3 1 1 1
input set 30 best modules are: 2 3 1 3 1
input set 31 best modules are: 3 3 2 2 2
input set 32 best modules are: 3 3 2 3 2
input set 33 best modules are: 3 3 2 2 2
input set 34 best modules are: 3 3 2 3 2
input set 35 best modules are: 2 1 2 3 2
input set 36 best modules are: 2 3 1 3 2
input set 37 best modules are: 2 1 2 3 2
input set 38 best modules are: 2 3 1 3 2
input set 39 best modules are: 3 3 1 3 1
input set 40 best modules are: 3 3 1 3 1
input set 41 best modules are: 3 3 1 3 1
input set 42 best modules are: 3 3 1 3 1
input set 43 best modules are: 2 1 1 3 2
input set 44 best modules are: 2 3 1 3 2
input set 45 best modules are: 2 1 1 3 2
input set 46 best modules are: 2 3 1 3 2
input set 47 best modules are: 3 3 1 3 1
input set 48 best modules are: 3 3 1 3 1
input set 49 best modules are: 3 3 1 3 1
input set 50 best modules are: 3 3 1 3 1

finding sub-tasks

module 2 of the first modular network
defines sub-task 1
module 3 of the first modular network
defines sub-task 2
so there are 2 sub-tasks

setting up a breeder for sub-task 1

input and desired output pattern 1 added
as training pattern
module 2 from modular network 1 added to
population
module 3 from modular network 2 added to
population
module 1 from modular network 3 added to
population
module 3 from modular network 4 added to
population
module 1 from modular network 5 added to
population
input and desired output pattern 2 added
as training pattern
module 2 from modular network 3 added to
population
input and desired output pattern 3 added
as training pattern
module 3 from modular network 3 added to
population
module 3 from modular network 5 added to
population
input and desired output pattern 4 added
as training pattern
input and desired output pattern 5 added
as training pattern
input and desired output pattern 6 added
as training pattern
input and desired output pattern 11 added
as training pattern
module 1 from modular network 4 added to
population
input and desired output pattern 12 added
as training pattern
input and desired output pattern 13 added
as training pattern
input and desired output pattern 14 added
as training pattern
input and desired output pattern 19 added
as training pattern
input and desired output pattern 20 added
as training pattern
input and desired output pattern 21 added
as training pattern
input and desired output pattern 22 added
as training pattern
input and desired output pattern 27 added
as training pattern
input and desired output pattern 28 added
as training pattern
input and desired output pattern 29 added
as training pattern
input and desired output pattern 30 added
as training pattern
input and desired output pattern 35 added
as training pattern
```

```
module 1 from modular network 2 added to
population
module 2 from modular network 5 added to
population
input and desired output pattern 36 added
as training pattern
input and desired output pattern 37 added
as training pattern
input and desired output pattern 38 added
as training pattern
input and desired output pattern 43 added
as training pattern
input and desired output pattern 44 added
as training pattern
input and desired output pattern 45 added
as training pattern
input and desired output pattern 46 added
as training pattern

setting up a breeder for sub-task 2

input and desired output pattern 7 added
as training pattern
module 3 from modular network 1 added to
population
module 3 from modular network 2 added to
population
module 2 from modular network 3 added to
population
module 3 from modular network 4 added to
population
module 2 from modular network 5 added to
population
input and desired output pattern 8 added
as training pattern
input and desired output pattern 9 added
as training pattern
module 2 from modular network 4 added to
population
input and desired output pattern 10 added
as training pattern
input and desired output pattern 15 added
as training pattern
module 1 from modular network 3 added to
population
input and desired output pattern 16 added
as training pattern
input and desired output pattern 17 added
as training pattern
input and desired output pattern 18 added
as training pattern
input and desired output pattern 23 added
as training pattern
input and desired output pattern 24 added
as training pattern
input and desired output pattern 25 added
as training pattern
input and desired output pattern 26 added
as training pattern
input and desired output pattern 31 added
as training pattern
input and desired output pattern 32 added
as training pattern
input and desired output pattern 33 added
as training pattern
input and desired output pattern 34 added
as training pattern
input and desired output pattern 39 added
as training pattern
module 1 from modular network 5 added to
population
input and desired output pattern 40 added
as training pattern
input and desired output pattern 41 added
as training pattern
input and desired output pattern 42 added
as training pattern
input and desired output pattern 47 added
as training pattern
input and desired output pattern 48 added
as training pattern
input and desired output pattern 49 added
as training pattern
input and desired output pattern 50 added
as training pattern

starting breeding


TEST: generation 1 stats

TEST: population size = 11
TEST: population member 1 has 0 nodes (0
active)
TEST: population member 2 has 4 nodes (4
active)
TEST: population member 3 has 3 nodes (3
active)
TEST: population member 4 has 0 nodes (0
active)
TEST: population member 5 has 8 nodes (8
active)
TEST: population member 6 has 1 nodes (1
active)
TEST: population member 7 has 1 nodes (1
active)
TEST: population member 8 has 2 nodes (2
active)
TEST: population member 9 has 3 nodes (3
active)
TEST: population member 10 has 1 nodes (1
active)
TEST: population member 11 has 6 nodes (6
active)
TEST: starting training population

TEST: starting training member 1
.....
.........
TEST: starting training member 2
.....
.........
TEST: starting training member 3
.....
.........
TEST: starting training member 4
.....
.........
TEST: starting training member 5
.....
.........
TEST: starting training member 6
.....
.........
TEST: starting training member 7
.....
.........
TEST: starting training member 8
.....
.........
TEST: starting training member 9
.....
.........
TEST: starting training member 10
.....
.........
TEST: starting training member 11
.....
.........
TEST: stopped training population
TEST: starting evaluating population
fitnesses
TEST: population member 1 error =
2.346548943012448
```

```
TEST: population member 1 fitness =
0.4261577424062683
TEST: population member 2 error =
6.517898790710071
TEST: population member 2 fitness =
0.15342367718647226
TEST: population member 3 error =
8.144048217244848
TEST: population member 3 fitness =
0.12278905690692271
TEST: population member 4 error =
2.5337925905126073
TEST: population member 4 fitness =
0.39466529491969654
TEST: population member 5 error =
14.66581536234601
TEST: population member 5 fitness =
0.06818577592129427
TEST: population member 6 error =
19.540320025086476
TEST: population member 6 fitness =
0.05117623450978124
TEST: population member 7 error =
16.506843266315066
TEST: population member 7 fitness =
0.060580935062288066
TEST: population member 8 error =
16.968475681267158
TEST: population member 8 fitness =
0.05893281275135274
TEST: population member 9 error =
12.462880751323373
TEST: population member 9 fitness =
0.0802382707460163
TEST: population member 10 error =
16.319407663943664
TEST: population member 10 fitness =
0.06127673384919567
TEST: population member 11 error =
27.703443196022896
TEST: population member 11 fitness =
0.036096596113495374
TEST: stopped evaluating population
fitnesses
TEST: member 1 is fittest

TEST: generating generation 2

TEST: number of members wanted to cross =
6
TEST: population members 7 4 selected for
crossover
TEST: child 1 added to new population as
member 1
TEST: child 2 added to new population as
member 2
TEST: population members 2 5 selected for
crossover
TEST: child 1 added to new population as
member 3
TEST: child 2 added to new population as
member 4
TEST: population members 1 9 selected for
crossover
TEST: child 1 added to new population as
member 5
TEST: child 2 added to new population as
member 6
TEST: number of members actually crossed =
6
TEST: 5 members to be kept unchanged
TEST: member 3 added to new population as
member 7
TEST: member 8 added to new population as
member 8
TEST: member 1 added to new population as
member 9
TEST: member 11 added to new population
as member 10
TEST: member 7 added to new population as
member 11
TEST: 6 members (from new population) to
be mutated
TEST: member 6 mutated
TEST: member 7 mutated
TEST: layer inserted at 0 with 1 nodes
TEST: member 5 mutated
TEST: layer inserted at 1 with 2 nodes
TEST: member 1 mutated
TEST: member 3 mutated
TEST: layer inserted at 1 with 2 nodes
TEST: member 4 mutated
TEST: layer inserted at 0 with 34 nodes

TEST: generation 2 stats

TEST: population size = 11
TEST: population member 1 has 0 nodes (0
active)
TEST: population member 2 has 1 nodes (1
active)
TEST: population member 3 has 8 nodes (8
active)
TEST: population member 4 has 6 nodes (6
active)
TEST: population member 5 has 5 nodes (5
active)
TEST: population member 6 has 0 nodes (0
active)
TEST: population member 7 has 3 nodes (3
active)
TEST: population member 8 has 3 nodes (3
active)
TEST: population member 9 has 34 nodes
(34 active)
TEST: population member 10 has 6 nodes (6
active)
TEST: population member 11 has 1 nodes (1
active)
TEST: starting training population

TEST: starting training member 1
.....
.........
TEST: starting training member 2
.....
.........
TEST: starting training member 3
.....
.........
TEST: starting training member 4
.....
.........
TEST: starting training member 5
.....
.........
TEST: starting training member 6
.....
.........
TEST: starting training member 7
.....
.........
TEST: starting training member 8
.....
.........
TEST: starting training member 9
.....
.........
TEST: starting training member 10
.....
.........
TEST: starting training member 11
.....
.........TEST: stopped training populat
```

```
ion
TEST: starting evaluating population
fitnesses
TEST: population member 1 error =
1.6395007597359927
TEST: population member 1 fitness =
0.6099417728608001
TEST: population member 2 error =
15.858206286967949
TEST: population member 2 fitness =
0.06305883413950707
TEST: population member 3 error =
13.5884502361773
TEST: population member 3 fitness =
0.07359190949808563
TEST: population member 4 error =
16.47975693760936
TEST: population member 4 fitness =
0.060680506623119246
TEST: population member 5 error =
14.685229923473488
TEST: population member 5 fitness =
0.06809563113489685
TEST: population member 6 error =
1.5592540839515663
TEST: population member 6 fitness =
0.6413322949045821
TEST: population member 7 error =
2.160705008280677
TEST: population member 7 fitness =
0.462811904525423
TEST: population member 8 error =
20.419016550626875
TEST: population member 8 fitness =
0.048973955112901826
TEST: population member 9 error =
25.816631521900433
TEST: population member 9 fitness =
0.038734720257818794
TEST: population member 10 error =
14.157323640600762
TEST: population member 10 fitness =
0.07063481950304311
TEST: population member 11 error =
15.858206286967949
TEST: population member 11 fitness =
0.06305883413950707
TEST: stopped evaluating population
fitnesses
TEST: member 6 is fittest
TEST: desired fitness reached

TEST: generation 1 stats

TEST: population size = 8
TEST: population member 1 has 0 nodes (0
active)
TEST: population member 2 has 4 nodes (4
active)
TEST: population member 3 has 1 nodes (1
active)
TEST: population member 4 has 0 nodes (0
active)
TEST: population member 5 has 6 nodes (6
active)
TEST: population member 6 has 6 nodes (6
active)
TEST: population member 7 has 3 nodes (3
active)
TEST: population member 8 has 8 nodes (8
active)
TEST: starting training population

TEST: starting training member 1
.....
.....
TEST: starting training member 2
.....
.....
TEST: starting training member 3
.....
.....
TEST: starting training member 4
.....
.....
TEST: starting training member 5
.....
.....
TEST: starting training member 6
.....
.....
TEST: starting training member 7
.....
.....
TEST: starting training member 8
.....
.....
TEST: stopped training population
TEST: starting evaluating population
fitnesses
TEST: population member 1 error =
2.068180349714895
TEST: population member 1 fitness =
0.483516826826951
TEST: population member 2 error =
11.270487659640729
TEST: population member 2 fitness =
0.08872730534819441
TEST: population member 3 error =
12.64265723847817
TEST: population member 3 fitness =
0.07909729585616548
TEST: population member 4 error =
2.270577018679533
TEST: population member 4 fitness =
0.44041668341272816
TEST: population member 5 error =
10.29069163178764
TEST: population member 5 fitness =
0.09717519830358436
TEST: population member 6 error =
26.238531257736096
TEST: population member 6 fitness =
0.03811188934994838
TEST: population member 7 error =
12.401509799875937
TEST: population member 7 fitness =
0.08063534328779903
TEST: population member 8 error =
6.703212271718719
TEST: population member 8 fitness =
0.14918220689788744
TEST: stopped evaluating population
fitnesses
TEST: member 1 is fittest

TEST: generating generation 2

TEST: number of members wanted to cross =
4
TEST: population members 5 1 selected for
crossover
TEST: child 1 added to new population as
member 1
TEST: child 2 added to new population as
member 2
TEST: population members 4 7 selected for
crossover
TEST: child 1 added to new population as
member 3
TEST: child 2 added to new population as
member 4
TEST: number of members actually crossed
= 4
```

```
TEST: 4 members to be kept unchanged
TEST: member 1 added to new population as
member 5
TEST: member 8 added to new population as
member 6
TEST: member 4 added to new population as
member 7
TEST: member 6 added to new population as
member 8
TEST: 4 members (from new population) to
be mutated
TEST: member 4 mutated
TEST: member 5 mutated
TEST: node 3 removed from layer 2
TEST: member 2 mutated
TEST: connection 3 removed from output 2
TEST: member 1 mutated

TEST: generation 2 stats

TEST: population size = 8
TEST: population member 1 has 0 nodes (0
active)
TEST: population member 2 has 6 nodes (6
active)
TEST: population member 3 has 3 nodes (3
active)
TEST: population member 4 has 0 nodes (0
active)
TEST: population member 5 has 0 nodes (0
active)
TEST: population member 6 has 7 nodes (7
active)
TEST: population member 7 has 0 nodes (0
active)
TEST: population member 8 has 6 nodes (6
active)
TEST: starting training population

TEST: starting training member 1
.....
.....
TEST: starting training member 2
.....
.....
TEST: starting training member 3
.....
.....
TEST: starting training member 4
.....
.....
TEST: starting training member 5
.....
.....
TEST: starting training member 6
.....
.....
TEST: starting training member 7
.....
.....
TEST: starting training member 8
.....
.....TEST: stopped training populat
ion
TEST: starting evaluating population
fitnesses
TEST: population member 1 error =
1.370242384324583
TEST: population member 1 fitness =
0.7297978893660613
TEST: population member 2 error =
13.266858121273074
TEST: population member 2 fitness =
0.07537579665501398
TEST: population member 3 error =
12.222886296452046
TEST: population member 3 fitness =
0.0818137366041171
TEST: population member 4 error =
1.455161876371046
TEST: population member 4 fitness =
0.6872087677927963
TEST: population member 5 error =
1.370242384324583
TEST: population member 5 fitness =
0.7297978893660613
TEST: population member 6 error =
12.152474417979121
TEST: population member 6 fitness =
0.0822877683676123
TEST: population member 7 error =
1.455161876371046
TEST: population member 7 fitness =
0.6872087677927963
TEST: population member 8 error =
26.22730347208864
TEST: population member 8 fitness =
0.038128204871088256
TEST: stopped evaluating population
fitnesses
TEST: member 1 is fittest
TEST: desired fitness reached

setting modular networks from breeders

taking members from breeder 1
population member 6 added as module in
modular network 1
population member 1 added as module in
modular network 2
population member 7 added as module in
modular network 3
population member 3 added as module in
modular network 4
population member 10 added as module in
modular network 5
taking members from breeder 2
population member 1 added as module in
modular network 1
population member 5 added as module in
modular network 2
population member 4 added as module in
modular network 3
population member 7 added as module in
modular network 4
population member 6 added as module in
modular network 5

finding best modular network

the error of modular network 1 =
0.15119914030554285
the error of modular network 2 =
0.10739070076224232
the error of modular network 3 =
0.1353945574954972
the error of modular network 4 =
0.1708024881067879
the error of modular network 5 =
0.25144689163660056
best modular network = 2

training all modular networks


TEST: training all modules and gating
network, on all examples (for 1 epoch per
example), for 50 epochs

m.....
.....
-EDITED-
.....
TEST: average error per epoch of module 1
```

```
= 0.06346829607325283
TEST: average weight per epoch of module 1
= 0.3882876310903775
TEST: average error per epoch of module 2
= 0.07568032669363639
TEST: average weight per epoch of module 2
= 0.3663402007494739

TEST: training all modules and gating
network, on all examples (for 1 epoch per
example), for 50 epochs

m.....................................
.....................................
-EDITED-
.........
TEST: average error per epoch of module 1
= 0.008956695295416189
TEST: average weight per epoch of module 1
= 0.416072675051528
TEST: average error per epoch of module 2
= 0.08575067360454604
TEST: average weight per epoch of module 2
= 0.2416748617554456

TEST: training all modules and gating
network, on all examples (for 1 epoch per
example), for 50 epochs

m.....................................
.....................................
-EDITED-
.........
TEST: average error per epoch of module 1
= 0.06621125679930567
TEST: average weight per epoch of module 1
= 0.4264438882887907
TEST: average error per epoch of module 2
= 0.022081959444451935
TEST: average weight per epoch of module 2
= 0.2990247137944984

TEST: training all modules and gating
network, on all examples (for 1 epoch per
example), for 50 epochs

m.....................................
.....................................
-EDITED-
.........
TEST: average error per epoch of module 1
= 0.27410735096473654
TEST: average weight per epoch of module 1
= 0.10882186324086639
TEST: average error per epoch of module 2
= 0.007186064153273938
TEST: average weight per epoch of module 2
= 0.7296107363617412

TEST: training all modules and gating
network, on all examples (for 1 epoch per
example), for 50 epochs

m.....................................
.....................................
-EDITED-
.........
TEST: average error per epoch of module 1
= 0.20434542089946361
TEST: average weight per epoch of module 1
= 0.28501589628080753
TEST: average error per epoch of module 2
= 0.2261209784751715
TEST: average weight per epoch of module 2
= 0.26803929014780775

setting best modules for each input
pattern

input set 1 best modules are: 1 1 2 2 1
input set 2 best modules are: 1 1 2 2 2
input set 3 best modules are: 1 1 1 2 1
input set 4 best modules are: 1 1 1 2 1
input set 5 best modules are: 1 1 1 2 1
input set 6 best modules are: 1 1 1 2 1
input set 7 best modules are: 2 1 2 2 2
input set 8 best modules are: 2 2 2 2 2
input set 9 best modules are: 2 2 2 2 2
input set 10 best modules are: 2 2 2 2 2
input set 11 best modules are: 1 1 1 2 2
input set 12 best modules are: 1 1 1 2 1
input set 13 best modules are: 1 1 1 2 1
input set 14 best modules are: 1 1 1 2 1
input set 15 best modules are: 2 2 2 2 2
input set 16 best modules are: 2 2 2 2 2
input set 17 best modules are: 1 2 2 2 2
input set 18 best modules are: 2 2 2 2 2
input set 19 best modules are: 2 1 1 2 2
input set 20 best modules are: 1 1 1 2 1
input set 21 best modules are: 1 1 1 2 2
input set 22 best modules are: 1 1 1 2 1
input set 23 best modules are: 1 2 2 2 2
input set 24 best modules are: 2 2 2 2 2
input set 25 best modules are: 2 2 2 2 2
input set 26 best modules are: 2 2 2 2 2
input set 27 best modules are: 2 1 1 2 2
input set 28 best modules are: 1 1 1 2 1
input set 29 best modules are: 2 1 1 2 2
input set 30 best modules are: 1 1 1 2 1
input set 31 best modules are: 1 2 2 2 2
input set 32 best modules are: 2 2 2 2 2
input set 33 best modules are: 1 2 2 2 2
input set 34 best modules are: 2 2 2 2 2
input set 35 best modules are: 1 1 1 2 1
input set 36 best modules are: 1 1 1 2 1
input set 37 best modules are: 1 1 1 2 1
input set 38 best modules are: 1 1 1 2 1
input set 39 best modules are: 2 2 2 2 2
input set 40 best modules are: 2 2 2 2 2
input set 41 best modules are: 2 2 2 2 1
input set 42 best modules are: 2 2 2 2 2
input set 43 best modules are: 1 1 1 2 1
input set 44 best modules are: 1 1 1 2 1
input set 45 best modules are: 1 1 1 2 1
input set 46 best modules are: 1 1 1 2 1
input set 47 best modules are: 2 2 2 2 1
input set 48 best modules are: 2 2 2 2 2
input set 49 best modules are: 2 2 2 2 1
input set 50 best modules are: 2 2 2 2 2

finding sub-tasks

module 1 of the first modular network
defines sub-task 1
module 2 of the first modular network
defines sub-task 2
so there are 2 sub-tasks

setting up a breeder for sub-task 1

input and desired output pattern 1 added
as training pattern
module 1 from modular network 1 added to
population
module 1 from modular network 2 added to
population
module 2 from modular network 3 added to
population
module 2 from modular network 4 added to
population
module 1 from modular network 5 added to
population
input and desired output pattern 2 added
as training pattern
```

```
module 2 from modular network 5 added to
population
input and desired output pattern 3 added
as training pattern
module 1 from modular network 3 added to
population
input and desired output pattern 4 added
as training pattern
input and desired output pattern 5 added
as training pattern
input and desired output pattern 6 added
as training pattern
input and desired output pattern 11 added
as training pattern
input and desired output pattern 12 added
as training pattern
input and desired output pattern 13 added
as training pattern
input and desired output pattern 14 added
as training pattern
input and desired output pattern 17 added
as training pattern
module 2 from modular network 2 added to
population
input and desired output pattern 20 added
as training pattern
input and desired output pattern 21 added
as training pattern
input and desired output pattern 22 added
as training pattern
input and desired output pattern 23 added
as training pattern
input and desired output pattern 28 added
as training pattern
input and desired output pattern 30 added
as training pattern
input and desired output pattern 31 added
as training pattern
input and desired output pattern 33 added
as training pattern
input and desired output pattern 35 added
as training pattern
input and desired output pattern 36 added
as training pattern
input and desired output pattern 37 added
as training pattern
input and desired output pattern 38 added
as training pattern
input and desired output pattern 43 added
as training pattern
input and desired output pattern 44 added
as training pattern
input and desired output pattern 45 added
as training pattern
input and desired output pattern 46 added
as training pattern

setting up a breeder for sub-task 2

input and desired output pattern 7 added
as training pattern
module 2 from modular network 1 added to
population
module 1 from modular network 2 added to
population
module 2 from modular network 3 added to
population
module 2 from modular network 4 added to
population
module 2 from modular network 5 added to
population
input and desired output pattern 8 added
as training pattern
module 2 from modular network 2 added to
population
input and desired output pattern 9 added
as training pattern
input and desired output pattern 10 added
as training pattern
input and desired output pattern 15 added
as training pattern
input and desired output pattern 16 added
as training pattern
input and desired output pattern 18 added
as training pattern
input and desired output pattern 19 added
as training pattern
module 1 from modular network 3 added to
population
input and desired output pattern 24 added
as training pattern
input and desired output pattern 25 added
as training pattern
input and desired output pattern 26 added
as training pattern
input and desired output pattern 27 added
as training pattern
input and desired output pattern 29 added
as training pattern
input and desired output pattern 32 added
as training pattern
input and desired output pattern 34 added
as training pattern
input and desired output pattern 39 added
as training pattern
input and desired output pattern 40 added
as training pattern
input and desired output pattern 41 added
as training pattern
module 1 from modular network 5 added to
population
input and desired output pattern 42 added
as training pattern
input and desired output pattern 47 added
as training pattern
input and desired output pattern 48 added
as training pattern
input and desired output pattern 49 added
as training pattern
input and desired output pattern 50 added
as training pattern

starting breeding


TEST: generation 1 stats

TEST: population size = 8
TEST: population member 1 has 0 nodes (0
active)
TEST: population member 2 has 0 nodes (0
active)
TEST: population member 3 has 0 nodes (0
active)
TEST: population member 4 has 0 nodes (0
active)
TEST: population member 5 has 6 nodes (6
active)
TEST: population member 6 has 7 nodes (7
active)
TEST: population member 7 has 3 nodes (3
active)
TEST: population member 8 has 0 nodes (0
active)
TEST: starting training population

TEST: starting training member 1
.....
.........
TEST: starting training member 2
.....
.........
TEST: starting training member 3
.....
```

```
.........
TEST: starting training member 4
.........
.........
TEST: starting training member 5
.........
.........
TEST: starting training member 6
.........
.........
TEST: starting training member 7
.........
.........
TEST: starting training member 8
.........
.........
TEST: stopped training population
TEST: starting evaluating population
fitnesses
TEST: population member 1 error =
1.3523475975575454
TEST: population member 1 fitness =
0.7394548574686604
TEST: population member 2 error =
1.1287202030578243
TEST: population member 2 fitness =
0.885959157363262
TEST: population member 3 error =
1.5681119470500533
TEST: population member 3 fitness =
0.6377095728919158
TEST: population member 4 error =
1.2712865802366433
TEST: population member 4 fitness =
0.7866047007385661
TEST: population member 5 error =
14.034910805868503
TEST: population member 5 fitness =
0.0712508981234041
TEST: population member 6 error =
15.452216914005213
TEST: population member 6 fitness =
0.06471563307486602
TEST: population member 7 error =
0.9571828247056348
TEST: population member 7 fitness =
1.044732494346138
TEST: population member 8 error =
2.301361722269734
TEST: population member 8 fitness =
0.43452534659077546
TEST: stopped evaluating population
fitnesses
TEST: member 7 is fittest
TEST: desired fitness reached

TEST: generation 1 stats

TEST: population size = 8
TEST: population member 1 has 0 nodes (0
active)
TEST: population member 2 has 0 nodes (0
active)
TEST: population member 3 has 0 nodes (0
active)
TEST: population member 4 has 0 nodes (0
active)
TEST: population member 5 has 7 nodes (7
active)
TEST: population member 6 has 0 nodes (0
active)
TEST: population member 7 has 3 nodes (3
active)
TEST: population member 8 has 6 nodes (6
active)
TEST: starting training population
TEST: starting training member 1
.........
.........
TEST: starting training member 2
.........
.........
TEST: starting training member 3
.........
.........
TEST: starting training member 4
.........
.........
TEST: starting training member 5
.........
.........
TEST: starting training member 6
.........
.........
TEST: starting training member 7
.........
.........
TEST: starting training member 8
.........
.........
TEST: stopped training population
TEST: starting evaluating population
fitnesses
TEST: population member 1 error =
1.131799873450384
TEST: population member 1 fitness =
0.8835484288855932
TEST: population member 2 error =
1.162418027681366
TEST: population member 2 fitness =
0.8602757150924994
TEST: population member 3 error =
0.9463779972273676
TEST: population member 3 fitness =
1.056660238223765
TEST: population member 4 error =
0.9038187612766001
TEST: population member 4 fitness =
1.1064165105264563
TEST: population member 5 error =
10.251427033112359
TEST: population member 5 fitness =
0.09754739479391265
TEST: population member 6 error =
1.2020396423173496
TEST: population member 6 fitness =
0.8319193184612049
TEST: population member 7 error =
1.6448340802542987
TEST: population member 7 fitness =
0.6079640566818725
TEST: population member 8 error =
17.529398885390552
TEST: population member 8 fitness =
0.05704702177970435
TEST: stopped evaluating population
fitnesses
TEST: member 4 is fittest
TEST: desired fitness reached

setting modular networks from breeders

taking members from breeder 1
population member 7 added as module in
modular network 1
population member 2 added as module in
modular network 2
population member 4 added as module in
modular network 3
population member 1 added as module in
modular network 4
population member 3 added as module in
modular network 5
```

```
taking members from breeder 2
population member 4 added as module in
modular network 1
population member 3 added as module in
modular network 2
population member 1 added as module in
modular network 3
population member 2 added as module in
modular network 4
population member 6 added as module in
modular network 5

finding best modular network

the error of modular network 1 =
0.058464622451071833
the error of modular network 2 =
0.013469816134839855
the error of modular network 3 =
0.014275987708537828
the error of modular network 4 =
0.01346197071701132
the error of modular network 5 =
0.018915693976082178
best modular network = 4

desired error reached


best modular network = 4


getting best network outputs on given
training input sets
input set 1
 network output 1 = 0.0162284976551846
 network output 2 = 0.9977200816342175
 network output 3 = 0.018181506802723013
input set 2
 network output 1 = 0.9630902368409311
 network output 2 = 0.9998128939879234
 network output 3 = 0.009081421954160644
input set 3
 network output 1 = 0.028150333279995565
 network output 2 = 0.024965566114520492
 network output 3 = 0.9938219256252514
input set 4
 network output 1 = 0.9513449067274232
 network output 2 = 0.00226658982360703
 network output 3 = 0.9994288066138943
input set 5
 network output 1 = 0.029670532809641742
 network output 2 = 0.02469759951234892
 network output 3 = 0.9937705608765763
input set 6
 network output 1 = 0.9513009511573634
 network output 2 = 0.0024655015827345435
 network output 3 = 0.9994208278175385
input set 7
 network output 1 = 0.024522802788504165
 network output 2 = 0.9461238733183384
 network output 3 = 0.9729197196336186
input set 8
 network output 1 = 0.6956861490580387
 network output 2 = 0.9889655010754879
 network output 3 = 0.9923756183130865
input set 9
 network output 1 = 0.0894043365483488
 network output 2 = 0.9440453828768163
 network output 3 = 0.9750854946223477
input set 10
 network output 1 = 0.8770445140533183
 network output 2 = 0.9995752040152623
 network output 3 = 0.9578334402152139
input set 11
 network output 1 = 0.042823990887475356
 network output 2 = 0.029301638589665883
 network output 3 = 0.9928287406774114
input set 12
 network output 1 = 0.9363457574092323
 network output 2 = 5.764802602204603E-4
 network output 3 = 0.9998056259870102
input set 13
 network output 1 = 0.049714505950558804
 network output 2 = 0.03027577003324982
 network output 3 = 0.9927066589068945
input set 14
 network output 1 = 0.9479298973573012
 network output 2 = 6.013659728527951E-4
 network output 3 = 0.999801467729392
input set 15
 network output 1 = 0.044955310756854966
 network output 2 = 0.947616306633368
 network output 3 = 0.9722561257261397
input set 16
 network output 1 = 0.7900414542427991
 network output 2 = 0.9978056173111497
 network output 3 = 0.9967055122619894
input set 17
 network output 1 = 0.10499122150240679
 network output 2 = 0.974978476265854
 network output 3 = 0.9852741741880262
input set 18
 network output 1 = 0.6542260824839334
 network output 2 = 0.9980235152461296
 network output 3 = 0.9989638735884054
input set 19
 network output 1 = 0.0942405373544062
 network output 2 = 0.02770356813752422
 network output 3 = 0.9945206228899381
input set 20
 network output 1 = 0.8570084639026765
 network output 2 = 8.321814112093504E-4
 network output 3 = 0.9999191133975684
input set 21
 network output 1 = 0.05537105939234297
 network output 2 = 0.0311954120512645
 network output 3 = 0.9943927150381132
input set 22
 network output 1 = 0.92249516730627
 network output 2 = 7.02420080347272E-4
 network output 3 = 0.99991653009567
input set 23
 network output 1 = 0.09004149490072828
 network output 2 = 0.9726511127325319
 network output 3 = 0.9822354426375395
input set 24
 network output 1 = 0.6895235197138868
 network output 2 = 0.9989189357004145
 network output 3 = 0.9984492122649435
input set 25
 network output 1 = 0.04756927458676506
 network output 2 = 0.9367552030628652
 network output 3 = 0.9738386046237291
input set 26
 network output 1 = 0.755073400944204
 network output 2 = 0.9983122426353566
 network output 3 = 0.9975723000146872
input set 27
 network output 1 = 0.10333727303318607
 network output 2 = 0.028678435289315185
 network output 3 = 0.9943703974172671
input set 28
 network output 1 = 0.8569924097828392
 network output 2 = 7.080442694536173E-4
 network output 3 = 0.9962994501494912
input set 29
 network output 1 = 0.13036507680580842
 network output 2 = 0.02840482077801061
 network output 3 = 0.9943217629931687
input set 30
 network output 1 = 0.9014761259325386
 network output 2 = 7.670954020429863E-4
 network output 3 = 0.9961865405221779
```

```
input set 31
 network output 1 = 0.09965437191033721
 network output 2 = 0.9722521625739213
 network output 3 = 0.9807302054919036
input set 32
 network output 1 = 0.669931275614264
 network output 2 = 0.998799214460974
 network output 3 = 0.9849944758405278
input set 33
 network output 1 = 0.1154608553394301
 network output 2 = 0.971847699805662
 network output 3 = 0.979781566212317
input set 34
 network output 1 = 0.6752859482874729
 network output 2 = 0.9979856108390339
 network output 3 = 0.9816000435881373
input set 35
 network output 1 = 0.041528787617437724
 network output 2 = 0.02449542514127141
 network output 3 = 0.016640945137490242
input set 36
 network output 1 = 0.9418034522875542
 network output 2 = 4.7446852151569627E-4
 network output 3 = 0.002764489376519178
input set 37
 network output 1 = 0.04139339830880391
 network output 2 = 0.024650604541099918
 network output 3 = 0.01674557867920211
input set 38
 network output 1 = 0.9424356999252707
 network output 2 = 4.778731929532075E-4
 network output 3 = 0.002821486231421061
input set 39
 network output 1 = 0.038965439880494224
 network output 2 = 0.9348219732593004
 network output 3 = 0.14380795756590292
input set 40
 network output 1 = 0.7639306047646819
 network output 2 = 0.9966396921824026
 network output 3 = 0.07692516989912078
input set 41
 network output 1 = 0.03885229339837172
 network output 2 = 0.9388448764656272
 network output 3 = 0.13787814614631336
input set 42
 network output 1 = 0.7415791896238111
 network output 2 = 0.9970282537589282
 network output 3 = 0.07465903847662694
input set 43
 network output 1 = 0.028051878764364626
 network output 2 = 0.025801207713009783
 network output 3 = 0.015673616719446427
input set 44
 network output 1 = 0.9534009116861502
 network output 2 = 0.0022295423874226276
 network output 3 = 3.361975232384451E-4
input set 45
 network output 1 = 0.0277153490984789
 network output 2 = 0.02590327351695322
 network output 3 = 0.015791381808173903
input set 46
 network output 1 = 0.9532265718410691
 network output 2 = 0.002223238537557193
 network output 3 = 3.1806108627839406E-4
input set 47
 network output 1 = 0.024700227821107396
 network output 2 = 0.9391101072360561
 network output 3 = 0.12230538055516832
input set 48
 network output 1 = 0.6728882438421318
 network output 2 = 0.9868531902295898
 network output 3 = 0.02048666424345231
input set 49
 network output 1 = 0.024042225472419347
 network output 2 = 0.9333718646718716
 network output 3 = 0.1276392397680674
input set 50
 network output 1 = 0.6119503031851801
 network output 2 = 0.9830350443406126
 network output 3 = 0.021311745463885468

getting network output given test
inputs...
test pattern 1;
 output 1 = 0.0162284976551846
 output 2 = 0.9977200816342175
 output 3 = 0.018181506802723013

test pattern 2;
 output 1 = 0.9630902368409311
 output 2 = 0.9998128939879234
 output 3 = 0.009081421954160644

test pattern 3;
 output 1 = 0.18995171368541053
 output 2 = 0.24765442778356217
 output 3 = 0.8811380667216923

test pattern 4;
 output 1 = 0.7560068728629108
 output 2 = 0.5261223789532891
 output 3 = 0.8258569215047351

test pattern 5;
 output 1 = 0.028150333279995565
 output 2 = 0.024965566114520492
 output 3 = 0.9938219256252514

test pattern 6;
 output 1 = 0.9513449067274232
 output 2 = 0.00226658982360703
 output 3 = 0.9994288066138943

test pattern 7;
 output 1 = 0.26020214656648477
 output 2 = 0.11091088508701172
 output 3 = 0.9749624889938575

test pattern 8;
 output 1 = 0.95049997218474
 output 2 = 0.002558812979308496
 output 3 = 0.9994324004577624

test pattern 9;
 output 1 = 0.029670532809641742
 output 2 = 0.02469759951234892
 output 3 = 0.9937705608765763

test pattern 10;
 output 1 = 0.9513009511573634
 output 2 = 0.0024655015827345435
 output 3 = 0.9994208278175385

test pattern 11;
 output 1 = 0.3030833049744916
 output 2 = 0.7603016934502447
 output 3 = 0.9552025849739314

test pattern 12;
 output 1 = 0.9843584472747533
 output 2 = 0.6434790953534031
 output 3 = 0.9978037941781541

test pattern 13;
 output 1 = 0.024522802788504165
 output 2 = 0.9461238733183384
 output 3 = 0.9729197196336186

test pattern 14;
 output 1 = 0.6956861490580387
 output 2 = 0.9889655010754879
 output 3 = 0.9923756183130865

test pattern 15;
```

```
 output 1 = 0.12004782571381425
 output 2 = 0.8969817101834668
 output 3 = 0.9714862107110385

test pattern 16;
 output 1 = 0.9225244292760293
 output 2 = 0.995128634142761
 output 3 = 0.9936808804849313

test pattern 17;
 output 1 = 0.0894043365483488
 output 2 = 0.9440453828768163
 output 3 = 0.9750854946223477

test pattern 18;
 output 1 = 0.8770445140533183
 output 2 = 0.9995752040152623
 output 3 = 0.9578334402152139

test pattern 19;
 output 1 = 0.0440030345968075
 output 2 = 0.79838141239054
 output 3 = 0.6600947834460815

test pattern 20;
 output 1 = 0.6986113926579585
 output 2 = 0.6633762450284648
 output 3 = 0.9889481459246046

test pattern 21;
 output 1 = 0.042823990887475356
 output 2 = 0.029301638589665883
 output 3 = 0.9928287406774114

test pattern 22;
 output 1 = 0.9363457574092323
 output 2 = 5.764802602204603E-4
 output 3 = 0.9998056259870102

test pattern 23;
 output 1 = 0.14496620423549336
 output 2 = 0.023505387740209063
 output 3 = 0.9929997649199992

test pattern 24;
 output 1 = 0.9376285808626195
 output 2 = 6.670666443108952E-4
 output 3 = 0.9998077756173553

test pattern 25;
 output 1 = 0.049714505950558804
 output 2 = 0.03027577003324982
 output 3 = 0.9927066589068945

test pattern 26;
 output 1 = 0.9479298973573012
 output 2 = 6.013659728527951E-4
 output 3 = 0.999801467729392

test pattern 27;
 output 1 = 0.18817162927291303
 output 2 = 0.4434815643685347
 output 3 = 0.9888787420355329

test pattern 28;
 output 1 = 0.9630001693127151
 output 2 = 0.39356525646334584
 output 3 = 0.999432506773251

test pattern 29;
 output 1 = 0.044955310756854966
 output 2 = 0.947616306633368
 output 3 = 0.9722561257261397

test pattern 30;
 output 1 = 0.7900414542427991
 output 2 = 0.9978056173111497
 output 3 = 0.9967055122619894

test pattern 31;
 output 1 = 0.0890397897437125
 output 2 = 0.9506506548800981
 output 3 = 0.9705927860728585

test pattern 32;
 output 1 = 0.7805503880292728
 output 2 = 0.9976650536636733
 output 3 = 0.9977600309079075

test pattern 33;
 output 1 = 0.10499122150240679
 output 2 = 0.974978476265854
 output 3 = 0.9852741741880262

test pattern 34;
 output 1 = 0.6542260824839334
 output 2 = 0.9980235152461296
 output 3 = 0.9989638735884054

test pattern 35;
 output 1 = 0.04811040430161197
 output 2 = 0.4064962509930949
 output 3 = 0.9969937652957634

test pattern 36;
 output 1 = 0.8299943671948133
 output 2 = 0.412136041433835
 output 3 = 0.999747484764399

test pattern 37;
 output 1 = 0.0942405373544062
 output 2 = 0.02770356813752422
 output 3 = 0.9945206228899381

test pattern 38;
 output 1 = 0.8570084639026765
 output 2 = 8.321814112093504E-4
 output 3 = 0.9999191133975684

test pattern 39;
 output 1 = 0.11414669220164754
 output 2 = 0.029282551222594176
 output 3 = 0.9959211896977441

test pattern 40;
 output 1 = 0.8861852478847063
 output 2 = 6.882377501582532E-4
 output 3 = 0.9999206759744904

test pattern 41;
 output 1 = 0.05537105939234297
 output 2 = 0.0311954120512645
 output 3 = 0.9943927150381132

test pattern 42;
 output 1 = 0.92249516730627
 output 2 = 7.02420080347272E-4
 output 3 = 0.99991653009567

test pattern 43;
 output 1 = 0.0679231603423466
 output 2 = 0.5078321659717369
 output 3 = 0.9908041365038227

test pattern 44;
 output 1 = 0.8578888213803469
 output 2 = 0.4826088100553547
 output 3 = 0.9997053439339743

test pattern 45;
 output 1 = 0.09004149490072828
 output 2 = 0.9726511127325319
 output 3 = 0.9822354426375395
```

```
test pattern 46;
 output 1 = 0.6895235197138868
 output 2 = 0.9989189357004145
 output 3 = 0.9984492122649435

test pattern 47;
 output 1 = 0.04823730172630288
 output 2 = 0.9618861098065231
 output 3 = 0.9816818507040456

test pattern 48;
 output 1 = 0.7377524717666863
 output 2 = 0.9983598591818328
 output 3 = 0.9978597992870786

test pattern 49;
 output 1 = 0.04756927458676506
 output 2 = 0.9367552030628652
 output 3 = 0.9738386046237291

test pattern 50;
 output 1 = 0.755073400944204
 output 2 = 0.9983122426353566
 output 3 = 0.9975723000146872

test pattern 51;
 output 1 = 0.0647233812581437
 output 2 = 0.5172245952215258
 output 3 = 0.585487136822956

test pattern 52;
 output 1 = 0.9075803801961009
 output 2 = 0.43912649826427624
 output 3 = 0.985041012421078

test pattern 53;
 output 1 = 0.10333727303318607
 output 2 = 0.028678435289315185
 output 3 = 0.9943703974172671

test pattern 54;
 output 1 = 0.8569924097828392
 output 2 = 7.080442694536173E-4
 output 3 = 0.9962994501494912

test pattern 55;
 output 1 = 0.08623499237743898
 output 2 = 0.02618012911365511
 output 3 = 0.85552604520493

test pattern 56;
 output 1 = 0.8600062841910219
 output 2 = 7.081397972250607E-4
 output 3 = 0.996378245388017

test pattern 57;
 output 1 = 0.13036507680580842
 output 2 = 0.02840482077801061
 output 3 = 0.9943217629931687

test pattern 58;
 output 1 = 0.9014761259325386
 output 2 = 7.670954020429863E-4
 output 3 = 0.9961865405221779

test pattern 59;
 output 1 = 0.05440880768463274
 output 2 = 0.46343970747526475
 output 3 = 0.9001793720883998

test pattern 60;
 output 1 = 0.8650289236473939
 output 2 = 0.4543377765644454
 output 3 = 0.9949983675985484

test pattern 61;
 output 1 = 0.09965437191033721
 output 2 = 0.9722521625739213
 output 3 = 0.9807302054919036

test pattern 62;
 output 1 = 0.669931275614264
 output 2 = 0.998799214460974
 output 3 = 0.9849944758405278

test pattern 63;
 output 1 = 0.04958469829883257
 output 2 = 0.9607466495559338
 output 3 = 0.8202575252790307

test pattern 64;
 output 1 = 0.6277793823493947
 output 2 = 0.998790620860744
 output 3 = 0.9844401381852763

test pattern 65;
 output 1 = 0.1154608553394301
 output 2 = 0.971847699805662
 output 3 = 0.979781566212317

test pattern 66;
 output 1 = 0.6752859482874729
 output 2 = 0.9979856108390339
 output 3 = 0.9816000435881373

test pattern 67;
 output 1 = 0.19168632590858412
 output 2 = 0.40367563516290006
 output 3 = 0.4247828640941451

test pattern 68;
 output 1 = 0.9508499595672987
 output 2 = 0.36213178498477405
 output 3 = 0.22166398710269505

test pattern 69;
 output 1 = 0.041528787617437724
 output 2 = 0.02449542514127141
 output 3 = 0.016640945137490242

test pattern 70;
 output 1 = 0.9418034522875542
 output 2 = 4.7446852151569627E-4
 output 3 = 0.002764489376519178

test pattern 71;
 output 1 = 0.19981561447497503
 output 2 = 0.02173967036310507
 output 3 = 0.3795727053155053

test pattern 72;
 output 1 = 0.9418575297548945
 output 2 = 4.6759717630711945E-4
 output 3 = 0.0028436940266040596

test pattern 73;
 output 1 = 0.04139339830880391
 output 2 = 0.024650604541099918
 output 3 = 0.01674557867920211

test pattern 74;
 output 1 = 0.9424356999252707
 output 2 = 4.778731929532075E-4
 output 3 = 0.002821486231421061

test pattern 75;
 output 1 = 0.07306466991351185
 output 2 = 0.5038886547659642
 output 3 = 0.5935371841702833

test pattern 76;
 output 1 = 0.8376588687238966
 output 2 = 0.40229330638809063
 output 3 = 0.020344182160976547
```

```
test pattern 77;
 output 1 = 0.038965439880494224
 output 2 = 0.9348219732593004
 output 3 = 0.14380795756590292

test pattern 78;
 output 1 = 0.7639306047646819
 output 2 = 0.9966396921824026
 output 3 = 0.07692516989912078

test pattern 79;
 output 1 = 0.13805396310084012
 output 2 = 0.9605968256292209
 output 3 = 0.5305433363827944

test pattern 80;
 output 1 = 0.7450469725135643
 output 2 = 0.9965690860306378
 output 3 = 0.08594127196587166

test pattern 81;
 output 1 = 0.03885229339837172
 output 2 = 0.9388448764656272
 output 3 = 0.13787814614631336

test pattern 82;
 output 1 = 0.7415791896238111
 output 2 = 0.9970282537589282
 output 3 = 0.07465903847662694

test pattern 83;
 output 1 = 0.29915256123450507
 output 2 = 0.7944478971682666
 output 3 = 0.8546416578670327

test pattern 84;
 output 1 = 0.9945001222519778
 output 2 = 0.70565518276443
 output 3 = 0.027679781889981027

test pattern 85;
 output 1 = 0.028051878764364626
 output 2 = 0.025801207713009783
 output 3 = 0.015673616719446427

test pattern 86;
 output 1 = 0.9534009116861502
 output 2 = 0.0022295423874226276
 output 3 = 3.361975232384451E-4

test pattern 87;
 output 1 = 0.2652441793738293
 output 2 = 0.09557840502102671
 output 3 = 0.07413725870620642

test pattern 88;
 output 1 = 0.9536983493356297
 output 2 = 0.0021517490477996988
 output 3 = 3.2410745223098505E-4

test pattern 89;
 output 1 = 0.0277153490984789
 output 2 = 0.02590327351695322
 output 3 = 0.015791381808173903

test pattern 90;
 output 1 = 0.9532265718410691
 output 2 = 0.002223238537557193
 output 3 = 3.1806108627839406E-4

test pattern 91;
 output 1 = 0.08062230384998545
 output 2 = 0.19812188391325436
 output 3 = 0.08004476988200943

test pattern 92;
 output 1 = 0.534875725900468
 output 2 = 0.21502694488738297
 output 3 = 0.0011328849247770703

test pattern 93;
 output 1 = 0.024700227821107396
 output 2 = 0.9391101072360561
 output 3 = 0.12230538055516832

test pattern 94;
 output 1 = 0.6728882438421318
 output 2 = 0.9868531902295898
 output 3 = 0.02048666424345231

test pattern 95;
 output 1 = 0.10437187999162718
 output 2 = 0.8453120456281022
 output 3 = 0.2555092834934636

test pattern 96;
 output 1 = 0.6302423871086705
 output 2 = 0.9831554907223627
 output 3 = 0.024368370412945782

test pattern 97;
 output 1 = 0.024042225472419347
 output 2 = 0.9333718646718716
 output 3 = 0.1276392397680674

test pattern 98;
 output 1 = 0.6119503031851801
 output 2 = 0.9830350443406126
 output 3 = 0.021311745463885468

test pattern 99;
 output 1 = 0.00557105204392507
 output 2 = 0.8048782553093347
 output 3 = 0.2936914501026188

test pattern 100;
 output 1 = 0.5554720894937253
 output 2 = 0.8924225706221214
 output 3 = 0.0690056008075765


getting network (difference) error given
test inputs...
test pattern 1;
 error = 0.03668992282369006

test pattern 2;
 error = 0.046178291125306165

test pattern 3;
 error = 1.8234353526235405

test pattern 4;
 error = 1.5437276696885354

test pattern 5;
 error = 0.05929397376926465

test pattern 6;
 error = 0.05149287648228948

test pattern 7;
 error = 0.396150542659639

test pattern 8;
 error = 0.052626440336806087

test pattern 9;
 error = 0.060597571445414326

test pattern 10;
 error = 0.0517437226078327
```

```
test pattern 11;
 error = 1.108182413450805

test pattern 12;
 error = 0.6613168539004957

test pattern 13;
 error = 0.10547920983654724

test pattern 14;
 error = 0.3229727315533869

test pattern 15;
 error = 0.2515799048193089

test pattern 16;
 error = 0.08866605609627842

test pattern 17;
 error = 0.17027345904918478

test pattern 18;
 error = 0.16554684171620548

test pattern 19;
 error = 1.1822896635412659

test pattern 20;
 error = 0.9758167064459018

test pattern 21;
 error = 0.07929688879972979

test pattern 22;
 error = 0.06442509686397797

test pattern 23;
 error = 0.17547182705570324

test pattern 24;
 error = 0.06323071016433608

test pattern 25;
 error = 0.08728361707691415

test pattern 26;
 error = 0.0528700008861596

test pattern 27;
 error = 0.6427744516059148

test pattern 28;
 error = 0.43113258037737967

test pattern 29;
 error = 0.1250828783973472

test pattern 30;
 error = 0.2154474161840617

test pattern 31;
 error = 0.1677963487907559

test pattern 32;
 error = 0.22402452739914636

test pattern 33;
 error = 0.1447385710485266

test pattern 34;
 error = 0.3487865286815315

test pattern 35;
 error = 0.4576128899989434

test pattern 36;
 error = 0.5823941894746226

test pattern 37;
 error = 0.12742348260199227

test pattern 38;
 error = 0.14390460411096442

test pattern 39;
 error = 0.14750805372649758

test pattern 40;
 error = 0.11458231389096152

test pattern 41;
 error = 0.09217375640549424

test pattern 42;
 error = 0.0782907226784072

test pattern 43;
 error = 0.5849511898102607

test pattern 44;
 error = 0.6250146447410335

test pattern 45;
 error = 0.13515493953065688

test pattern 46;
 error = 0.3131083323207552

test pattern 47;
 error = 0.10466934121573421

test pattern 48;
 error = 0.2660278697644023

test pattern 49;
 error = 0.13697546690017082

test pattern 50;
 error = 0.24904205640575217

test pattern 51;
 error = 0.9964608396567135

test pattern 52;
 error = 0.5465051056470973

test pattern 53;
 error = 0.13764531090523419

test pattern 54;
 error = 0.14741618433712325

test pattern 55;
 error = 0.2568890762861641

test pattern 56;
 error = 0.14432361021818618

test pattern 57;
 error = 0.16444813459065039

test pattern 58;
 error = 0.1031044289473265

test pattern 59;
 error = 0.6176691430714977

test pattern 60;
 error = 0.5943104853185031

test pattern 61;
 error = 0.1466720038445123

test pattern 62;
```

```
 error = 0.3462750340842342

test pattern 63;
 error = 0.26858052346386807

test pattern 64;
 error = 0.388989858604585

test pattern 65;
 error = 0.16383158932145106

test pattern 66;
 error = 0.3451283972853558

test pattern 67;
 error = 1.0201448251656293

test pattern 68;
 error = 0.6329458125201705

test pattern 69;
 error = 0.08266515789619938

test pattern 70;
 error = 0.06143550561048063

test pattern 71;
 error = 0.6011279901535854

test pattern 72;
 error = 0.06145376144801673

test pattern 73;
 error = 0.08278958152910594

test pattern 74;
 error = 0.06086365949910361

test pattern 75;
 error = 1.1704905088497592

test pattern 76;
 error = 0.5849786198251706

test pattern 77;
 error = 0.24795142418709676

test pattern 78;
 error = 0.31635487295203635

test pattern 79;
 error = 0.7080004738544137

test pattern 80;
 error = 0.34432521342166955

test pattern 81;
 error = 0.2378855630790579

test pattern 82;
 error = 0.3360515950938876

test pattern 83;
 error = 1.9482421162698045

test pattern 84;
 error = 0.7388348424024331

test pattern 85;
 error = 0.06952670319682083

test pattern 86;
 error = 0.04916482822451092

test pattern 87;
 error = 0.43495984310106245

test pattern 88;
 error = 0.04877750716440099

test pattern 89;
 error = 0.06941000442360602

test pattern 90;
 error = 0.04931472778276652

test pattern 91;
 error = 0.35878895764524926

test pattern 92;
 error = 0.681284103911692

test pattern 93;
 error = 0.20789550114021957

test pattern 94;
 error = 0.3607452301717307

test pattern 95;
 error = 0.5145691178569884

test pattern 96;
 error = 0.4109704925819125

test pattern 97;
 error = 0.21830960056861515

test pattern 98;
 error = 0.4263263979380928

test pattern 99;
 error = 1.1041407574558786

test pattern 100;
 error = 1.4059560819359727

 average error = 0.3715018660531947


end

->
```

APPENDIX C – Test Parameters

1. Neural Network

1) Test ww1-network-2:
2) Test ww1-network-3:

backpropagation step constant = 0.1
number of epochs = 1000

2. Modular Network

1) Test ww1-modular-4:

expert backpropagation step constant = 0.1
training algorithm = winner-takes-all
step constant = 0.1
memory constant = 0.1
improvement constant = 1.0
number of epochs = 1000

3. Genetic Algorithm

1) Test ww1-breeder-1:

size of population = 6
max. number of layers = 3
max. number of nodes per layer = 5
fitness evaluator = LinearEvaluator
selection algorithm = FitnessProportionate
crossover operation = CrossoverOperation1
mutation algorithm = MutateAll
number of epochs of training before evaluation = 200
fitness threshold = 0.95
percentage replaced = 50
percentage mutated = 10

2) Test ww1-breeder-4:

size of population = 6
max. number of layers = 3
max. number of nodes per layer = 5
fitness evalutaor = SquareEvaluator
selection algorithm = FitnessProportionate
crossover operation = CrossoverOperation1
mutation algorithm = MutateNodes

number of epochs of training before evaluation = 100
fitness threshold = 0.9
percentage replaced = 50
percentage mutated = 50

4. Modular Breeder

number of modular networks = 5
desired error = 0.1
percentage improvement that breeder must achieve = 30
module training step constant = 0.1
modular network training memory constant = 0.1
modular network training step constant = 0.1
modular network training improvement constant = 1.0
percentage of population to replace = 50

1) Test ww1-modbreeder-3:

initial number of modules in each network = 3
maximum number of layers in randomly generated networks = 3
maximum number of nodes per layer in randomly generated networks = 6
number of epochs to train modular networks = 100
number of epochs to train each population member = 100
percentage of population to mutate = 20

2) Test ww1-modbreeder-4

initial number of modules in each network = 3
maximum number of layers in randomly generated networks = 3
maximum number of nodes per layer in randomly generated networks = 6
number of epochs to train modular networks = 100
number of epochs to train each population member = 100
percentage of population to mutate = 20

3) Test ww1-modbreeder-6

initial number of modules in each network = 3
maximum number of layers in randomly generated networks = 2
maximum number of nodes per layer in randomly generated networks = 5
number of epochs to train modular networks = 50
number of epochs to train each population member = 50
percentage of population to mutate = 50