An Evolutionary Approach to Training Feed-Forward and Recurrent Neural Networks

by

Jeff Riley

A minor thesis submitted in partial fulfilment of the requirements for the degree of

Master of Applied Science in Information Technology

Department of Computer Science

Royal Melbourne Institute of Technology
Australia

November 1997

Declaration

I certify that that all work on this thesis was carried out between February 1997 and November 1997, and it has not been submitted for any academic award at any other college, institute or university. The work presented was carried out under the supervision of Dr. Victor B. Ciesielski of the Department of Computer Science, Royal Melbourne Institute of Technology. All work in the thesis is my own except where acknowledged in the text.

Jeff Riley
November 27, 1997

Acknowledgments

Thanks must go to my wife Jennifer for her unfaltering support throughout the years that have culminated in this thesis. Without her support, patience, and constant encouragement, this thesis would not have been possible.

I thank also my supervisor, Dr. Vic. Ciesielski whose advice and teaching over the entire course were invaluable.

Thanks also to Karen, whose continued encouragement led me to embark on this journey.

Finally, thankyou to my parents, whose faith and encouragement got me to this place.

Abstract

Artificial neural networks exhibit many useful and unique attributes: their ability to approximate and generalise chief amongst them. Feed-forward neural networks are very good at recognising patterns in noisy data, and determining relationships and mappings between data. Recurrent neural networks are powerful tools for solving problems with a temporal component, such as time series prediction, speech recognition and real time control systems. The variations of the gradient descent techniques for training both feed-forward and recurrent neural networks are computationally expensive and time consuming, and not always able to find a good solution. In short, neural networks, particularly recurrent neural networks, can be difficult and expensive to train.

Evolutionary methods such as genetic algorithms are global search techniques and as such are less likely to be fooled by local variations in the error landscape. This thesis investigates the possibility of using an evolutionary approach to train feed-forward and recurrent neural networks.

The technique investigated by this thesis is the use of a genetic algorithm to evolve changes to the weights and biases of the neural network, rather than evolve the weights and biases directly. The structure of the gene used by this technique obviates the need for real values to be encoded on the chromosome.

This thesis tests the technique on a number of problems for both feed-forward and recurrent neural networks. Standard parity, encoder and converter problems are tested for feed-forward networks, and sequence generation and time series prediction problems tested for recurrent networks. Several different recurrent architectures are tested, including simple recurrent networks and real time recurrent networks. A method of stopping training to prevent over-training is also investigated.

Results attained by the tests conducted indicate that the technique developed as part of this work is capable of training both feed-forward and recurrent neural networks, and that it compares favourably with, or is superior to, gradient descent techniques in some cases.

The technique developed for this work has shown to be very promising, for both feed-forward and recurrent neural networks, and when used in conjunction with early stopping techniques can overcome some of the problems associated with gradient descent methods. Further investigation into the determination of optimum control parameters for the technique is necessary to improve the performance of the technique.

Contents

Figures

Tables

Chapter 1
Introduction

1.1 Introduction

Artificial neural networks have applications in many areas. Feed-forward networks are particularly useful for input-output mapping and classification problems. Recurrent networks have uses in time series prediction, grammar learning, real-time control systems, pattern and speech recognition, etc., and provide significant advantages over feed-forward networks. Although useful, artificial neural networks, particularly recurrent neural networks, are difficult to train and so have not been utilised to their full potential.

The most common techniques for training both feed-forward and recurrent neural networks are variations of the gradient descent technique. These gradient descent techniques suffer from well-known problems, so more efficient methods to determine network weights are being sought. One such method combines another biologically inspired technique, that of genetic algorithms, with neural networks.

There are several advantages to using genetic algorithms over the gradient descent techniques. Two of the most important are that they require no knowledge about the response surface, including gradient information, and that they are far less likely to become trapped in a local minimum.

Several variations of the genetic algorithm-neural network hybrid exist; the most common of which are the determination of network weights by the use of genetic algorithms, and the evolutionary design of the network architecture. This work investigates a novel method of encoding network weights and biases onto a chromosome to allow a genetic algorithm to determine the weights and biases of a fixed architecture artificial neural network. This method, known as 2DELTA-GANN, is described in detail in chapter 3. A brief description follows.

2DELTA-GANN is a technique developed by Rajendra Krishnan and described in [Krishnan, 1994a] and [Krishnan, 1994b] which uses a genetic algorithm to train fixed architecture feed-forward neural networks. The approach used in 2DELTA-GANN is to have the genetic algorithm evolve some change, or delta value, to the weights of the neural network being trained. This is done by modifying the weights and biases of the network by some value according to some combination of fixed rules. Rather than have the genetic algorithm evolve the actual weights and biases of the network, only the way in which the rules are to be applied and the delta values are evolved. As a result, the chromosome being modified by the genetic algorithm does not need to represent each weight and bias of the network; only the method by which the rules are to be applied and the delta values need be represented.

The gene structure used by Krishnan is an attempt to overcome the problems associated with encoding real numbers onto a chromosome represented as a bit string. In the 2DELTA-GANN method, each gene on the chromosome is a composite structure representing either a weight or a bias from the neural network. The gene is composed of three rule bits and two floating point values. The floating point values are manipulated in accordance with the rule bits to apply a change to the weights and biases of the network. The three rule bits are

denoted ***x1***, ***x2*** and ***x3***; and the two floating point values are denoted ***delta1*** and ***delta2***. The basic concept of the 2DELTA-GANN method is to use the rule bits to specify a simple heuristic to apply to the ***delta2*** value, which is then used to modify the value of ***delta1***. Finally, ***delta1*** is used to modify the network weight or bias associated with the gene.

The machine learning community has reported some success in training feed-forward neural networks by the use of genetic algorithms, and little success in training recurrent neural networks by the use of genetic algorithms. The 2DELTA-GANN method shows that some success can be had when applying genetic algorithms to evolve changes in the weights of simple feed-forward neural networks. This work verifies and improves the 2DELTA-GANN technique with regard to training feed-forward neural networks, and extends the technique by applying it to recurrent neural networks.

1.2 Goals

The overall goal of this research is to investigate the viability of utilising genetic algorithms to determine the network weights and biases for fixed architecture feed-forward and recurrent neural networks, and to compare this method with existing, more common methods for training artificial neural networks.

The specific goals are:

- To test, verify, and where possible improve Krishnan's original implementation of the 2DELTA-GANN method of utilising genetic algorithms for training feed-forward neural networks.

- To test the hypothesis that the evolution of the rules governing the modification of the delta values and their application to the weights and biases of the network provides little or no benefit to the overall 2DELTA-GANN algorithm, and in so doing show that the worth of the technique is in the usefulness of genetic recombination as it is applied to the delta values themselves.

- To investigate the possibility of extending the technique to include recurrent neural networks, and attempt to develop the method to the point that it compares favourably with the more commonly used gradient descent techniques. To facilitate this several variations of the 2DELTA-GANN method will be developed, and results obtained by these new methods will be compared and contrasted with results obtained by the original 2DELTA-GANN, as well as with results obtained by the more common gradient descent techniques for training neural networks.

- To investigate and evaluate a method of stopping training to prevent over-fitting by the 2DELTA-GANN technique. The method to be investigated uses a validation data set, different from the training data set, to monitor the performance of the network being evolved. After each generation the best performing individual is evaluated using the validation data set, and if the network error for the validation data set increases from the minimum by some arbitrary amount, training is stopped.

1.3 Thesis Content

The early chapters of this thesis present a literature review including a brief introduction to and description of genetic algorithms and neural networks, and a discussion of the 2DELTA-GANN method of using genetic algorithms to train feed-forward neural networks.

The verification and improvements to the method for training feed-forward neural networks are then described, and several feed-forward networks are trained for comparison with the original 2DELTA-GANN results. These networks are the XOR network, a 4-2-4 encoder network, and a digital to analogue converter network

An analysis of the 2DELTA-GANN technique, and a description of modifications to the method are then presented, followed by results of the tests run to determine the viability of training recurrent neural networks by this method. Several recurrent networks of different architectures are trained by the new 2DELTA-GANN method, and the results compared with gradient descent techniques for the same networks. Both simple recurrent networks and real time recurrent networks are trained for various problems, including sequence generation and time series prediction.

An investigation into the use of a method of stopping training in order to prevent over-fitting is also presented. This method is evaluated using a recurrent neural network to predict sunspot activity.

An analysis of the results and conclusions drawn are also presented. Some possible further work based on the results of this work is also described.

Chapter 2
Literature Review

2.1 Genetic Algorithms

Developed by John Holland [Holland, 1975], a genetic algorithm is a biologically inspired search technique. In simple terms, the technique involves generating a random initial population of individuals, each of which represents a potential solution to a problem. Each member of that population's fitness as a solution to the problem is evaluated against some known criteria. Members of the population are then selected for reproduction based upon that fitness, and a new generation of potential solutions is generated from the offspring of the most fit individuals. The process of evaluation, selection, and recombination is iterated until the population converges to an acceptable solution.

The evaluation of an individual's worth as a solution is achieved by the use of a fitness function. The goal of the fitness function is to numerically encode the performance of the individual with reference to the problem for which it is a potential solution. This is an extremely important process, for without a fitness function which accurately evaluates the performance of potential solutions, the search will fail.

Selection of individuals which will have some part of their genetic material propagated through to the next generation of potential solutions is achieved by using one of several different selection methods. Probably the most common method of selection can be likened to a biased roulette wheel [Goldberg, 1989], where each individual in the current population has a slot on the roulette wheel proportional to that individual's fitness. The roulette wheel is spun once for each parent required, with the winning individuals being paired for reproduction. Since by this method individuals with a low fitness still have a chance, albeit small, of being selected for reproduction, the diversity of the population is to some extent retained.

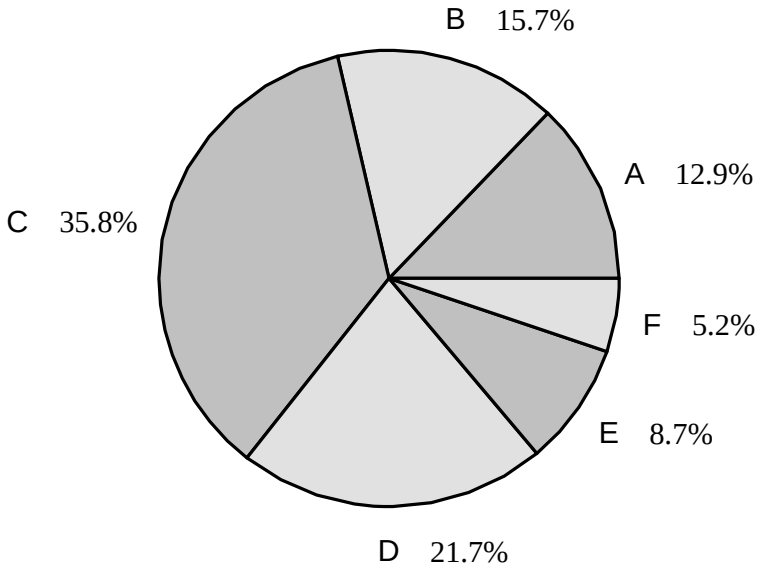


Figure 1 Example of biased roulette wheel for selection.

Figure 1 shows an example of a biased roulette wheel, showing a population of six individuals and their corresponding proportion of the roulette wheel based upon their fitness.

Recombination is achieved by the use of one of a number of recombination operators. Central to Holland's genetic algorithm is the implementation of the crossover recombination operator. In earlier works, evolutionary algorithms relied solely on mutation of individuals in the population of potential solutions in order to evolve a good solution. The crossover operator approximates sexual reproduction in biology, and is the combination of genetic material from two individuals to produce one or more offspring. Potential solutions are encoded onto chromosomes, usually bit strings of some arbitrary length, and those chromosomes are the genetic material manipulated by the crossover and mutation operators.

2.1.1 The Schema Theorem

Fundamental to understanding the search heuristics of genetic algorithms is Holland's *Schema Theorem* [Holland, 1975] and the concept of *schemata*, or *similarity templates* [Goldberg, 1989]. In terms of bit strings defined over the alphabet $\{0,1\}$, a schema is simply a template string defined over the expanded alphabet $\{0,1,*\}$, where * is a wild-card or don't-care character. Using this extended alphabet, bit strings can be said to be instances of many different similarity templates, or schemata.

For example, the bit string

010

is an instance of each schema in the set

$\{***,0**,*1*,**0,01*,0*0,*10,010\}$,

and conversely the schema

$00*1*$

describes the set of bit strings $\{00010,00110,00011,00111\}$

For any schema H, the order of H, $o(H)$, is defined by Holland to be the number of defined bits in the schema. Holland further defines the defining length of H, $d(H)$, to be the distance (number of bits) between the first and last defined bits of H.

For example, the schema

$0*01*011**$

has $o(H) = 6$ and $d(H) = 7$.

An important insight provided by Holland is that as the genetic algorithm explicitly searches through bit strings defined over the alphabet ***{0,1}***, it *implicitly* searches the much larger space of schemata defined over the alphabet ***{0,1,*}***. That is, knowledge of the fitness of a particular bit string implies some knowledge of the fitness of *all* the schemata of which that bit string is a member. Thus, the genetic algorithm can be said to be inherently parallel and so enable very large solution spaces to be searched in a reasonable time frame.

Holland derived a mathematical expression which predicts the number of copies of a given schema expected in a population after undergoing selection, crossover and mutation. Analysis of this expression suggests that schemata of lower order and defining length have a greater probability of survival after undergoing selection, crossover and mutation.

The Schema Theorem described by Holland states that the expected number of copies of very fit, short (in terms of defining length), low order schemata increases exponentially. Conversely, the expected number of copies of short, low order schemata of low fitness decreases exponentially. The Building Block Hypothesis [Goldberg, 1989] further states that the propagation and recombination from generation to generation of small, highly fit schemata is fundamental to the power of genetic algorithms. It is this recombination of short, low order, highly fit schemata forming even more highly fit higher order schemata which gives the genetic algorithm its power.

2.1.2 Crossover and Mutation

Many crossover operators have been developed by genetic algorithm practitioners. Single point, two point, and uniform crossover are just three examples. Single point crossover is achieved by randomly choosing a single point at which to separate, swap, and rejoin the bit string. Figure 2 is an example of single point crossover. Two point crossover is an extension of single point crossover. In two point crossover, two points are chosen at random and the segment of the bit string between those points is exchanged.

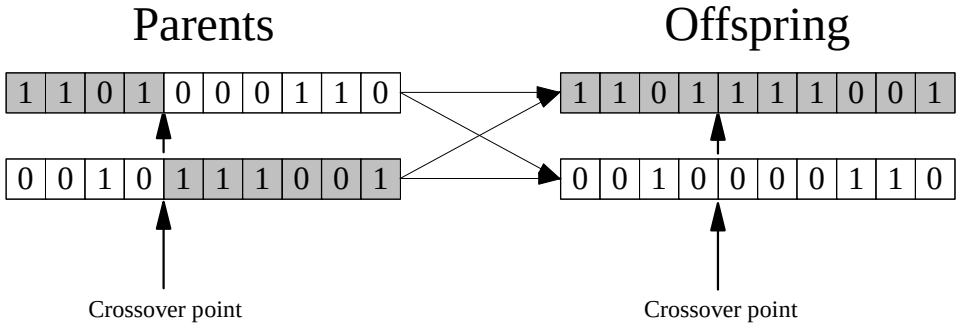


Figure 2 Example of single point crossover.

Single point crossover suffers from several problems. Arguably the most serious of these problems is the propensity for schemata with long defining lengths to be destroyed when the chromosome of which they are part undergoes single point crossover. A similarly serious problem for single point crossover is its inability in some circumstances to combine all possible schemata. For example, single point crossover is not able to combine the parent schemata ***1*11***1** and ***1****11**** to produce the offspring ***11*1111*1***. This

phenomenon is known as positional bias because the schemata which can be created or destroyed depend largely on the location of the bits on the chromosome. Two point crossover is less likely to destroy schemata with large defining lengths, and while it also is unable to combine all possible schemata, it can combine more than single point crossover.

Uniform crossover is implemented by generating a bit mask equal in length to the chromosome being manipulated, with the value of each bit being determined with some arbitrary probability. For each bit of the mask which is on (or a "*1*") the corresponding bit of the parent chromosomes is swapped (or crossed over) before propagation to the offspring; and for each mask bit which is off (or a "*0*") the corresponding parent bits are propagated to the offspring unchanged. Figure 3 is an example of uniform crossover.

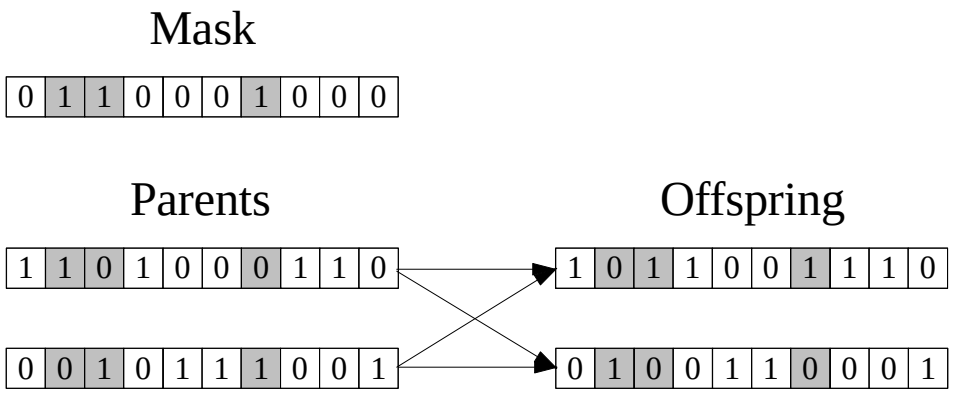


Figure 3 Example of uniform crossover.

Because each bit on the chromosome has some probability of being exchanged, uniform crossover suffers from no positional bias. Uniform crossover is able to combine all possible schemata, but it can be extremely disruptive of schemata of any length.

Mutation, which helps to maintain diversity in the population, is the random modification of individuals. Figure 4 shows an example of a single bit mutation.

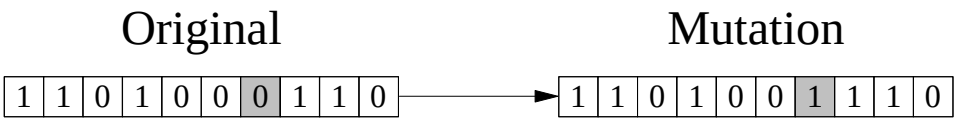


Figure 4 Example of single bit mutation.

2.2 Artificial Neural Networks

McCulloch and Pitts, in [McCulloch, 1943], introduced a model of a biological neuron and described a logical calculus of neural networks. Later work by Rosenblatt [Rosenblatt, 1962], in which the *perceptron* was defined, and Rumelhart [Rumelhart, 1986] extended the basic model neuron described by McCulloch and Pitts. An example of the McCulloch-Pitts neuron is shown in figure 5.

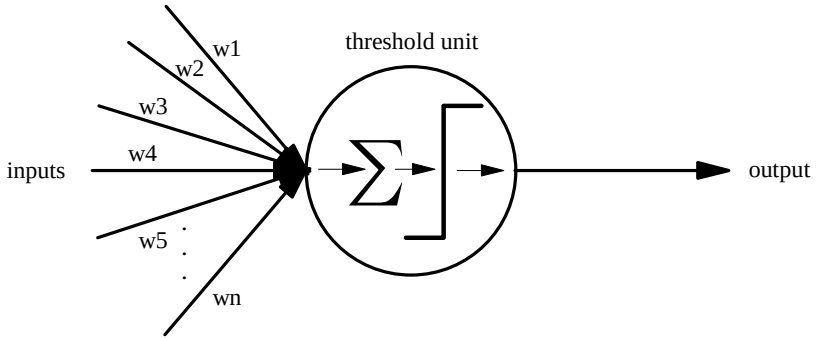


Figure 5 The McCulloch-Pitts neuron.

The model of the neuron proposed by McCulloch and Pitts is implemented as a threshold unit. Weighted inputs to the unit are summed to produce an activation signal, and if the activation signal exceeds some threshold value the unit produces some output response. If the activation signal does not exceed the threshold, no output is produced by the unit. Suppose there are n inputs to the threshold unit with weights $w_1, w_2, \ldots w_n$ and signals $x_1, x_2, \ldots x_n$. The activation α of the unit is then

$$\alpha = \sum_{i=1}^{n} w_i x_i$$

The output of the threshold unit is given by

$$o = \begin{cases} 1 \ \textit{if} \ \alpha \geq \theta \\ 0 \ \textit{if} \ \alpha < \theta \end{cases}$$

where θ is the threshold and often equal to zero.

The threshold unit works with binary signals only. To cater for continuous, or real-valued, input and output signals, other activation functions can be used. The simplest of these is a linear activation function which just sums the inputs to the neuron and uses that sum of the inputs as the neuron's output. Another approach is to soften the step activation function of the threshold unit slightly by implementing a sigmoidal activation function. The output of such a sigmoidal unit is given by

$$o = \frac{1}{1 + e^{\frac{-\alpha}{\rho}}}$$

where α is the summed input to the unit, and ρ is often assumed to be 1 and omitted.

Artificial Neural Networks (ANNs) are networks of interconnected neurons which are intended to model the biological neural networks of the brain. The basic construction of ANNs is such that an input vector presented at the input units causes the ANN to produce the correct, or expected, output vector at the output units. The complexity of the connections, and the inherent redundancy of the network allows ANNs to demonstrate good learning ability, generalisation and robustness.

Since the perceptron was described by Rosenblatt in 1953 there have been many different ANN techniques and architectures developed with different characteristics and capabilities.

ANNs are generally arranged in layers, with each layer having a specific purpose or performing a specific function. The layer to which the inputs are presented is referred to as the input layer, and the layer from which the output of the network is taken is referred to as the output layer. Between the input and output layers are very often one or more layers referred to as hidden layers. The hidden layers act as transducers between the layers to which they are connected.

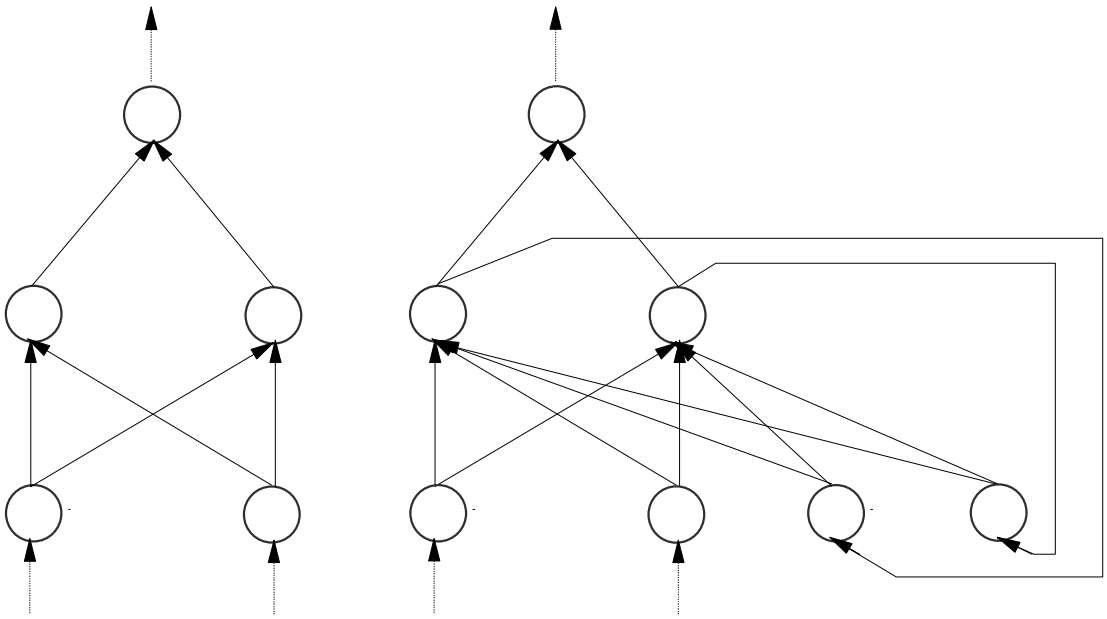


Figure 6 Feed-forward and recurrent networks.

There are essentially two architectures for ANNs: feed-forward networks and recurrent networks. Figure 6 shows simple feed-forward and recurrent networks. Both feed-forward and recurrent networks are composed of interconnected units, or neurons, usually arranged in layers. In feed-forward networks, activation signals move in one direction: from the input layer to the output layer. In recurrent networks, some activation signals travel back to a previous layer and become input signals to the units on that previous layer. These connections are said to be recurrent connections, with the units and layers often referred to as context units and layers.

2.2.1 Recurrent Networks

Recurrent neural networks have applications in many areas which require temporal processing, including time series prediction, grammar learning, real-time control systems, pattern and speech recognition etc., and provide significant advantages over feed-forward networks.

Recurrent networks are constructed to include feedback, or recurrent, connections to a secondary set of input, or context, units (see figure 6). The recurrent connections allow the network to store activity patterns and present those patterns to the network more than once. If each presentation of an input pattern is considered a time step in some time series, then the recurrent connections allow activity patterns to be presented again to the network at some later time step. For any given time step, the network output is calculated by propagating the input pattern forward through the network; and the recurrent activations are propagated back to the context units for presentation as inputs at some subsequent time step.

Many recurrent architectures and learning algorithms exist. Some of the more common are Back-Propagation Through Time, Recurrent Back-Propagation for Simple Recurrent Networks, and Real Time Recurrent Learning for Real Time Recurrent Networks.

2.2.1.1 Back-Propagation Through Time

Back-Propagation Through Time (BPTT) [Werbos, 1974 and Rumelhart, 1986] involves unfolding the temporal operation of a fully connected recurrent network into a multilayer feed-forward network. This results in a larger network with a new set of layers for each time step, but with the same weights and biases used for each time step. An example of this technique is shown pictorially in figures 7 and 8. Figure 7 shows the original recurrent network, consisting of three units, each with a modifiable bias and modifiable connection to itself and each other unit. Figure 8 shows the network unfolded through two time steps. The bias values for the corresponding hidden and output units are identical for each time step, as are the input weights to the corresponding hidden and output units.

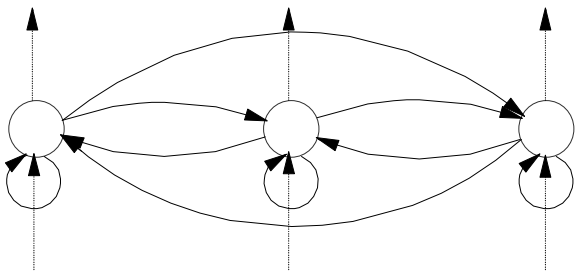


Figure 7 Fully connected recurrent network.

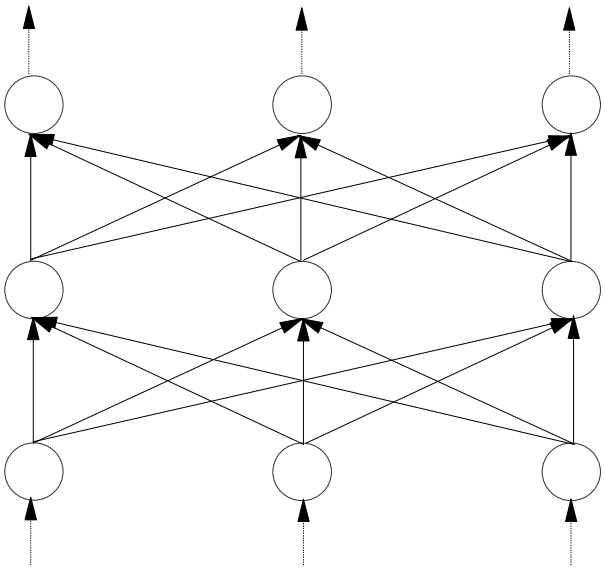


Figure 8 Recurrent network unfolded through time.

2.2.1.2 Simple Recurrent Networks

Simple Recurrent Networks (SRN) [Elman, 1991] introduce recurrence to the network by the addition of a set of units called *context units*. Simple recurrent networks have the same number of context units as main hidden units with the context units being activated on a one-for-one basis by the main hidden units. Elman notes that additional hidden layers between the input layer and the main hidden layer, or between the main hidden layer and the output layer may be used to compress the input and output vectors. Context units have a fixed weight of *1.0*. With simple recurrent networks, the network state at any time is a function of the input at the current time step plus the state of the main hidden units at the previous time step. Figure 9 is an example of a simple recurrent network.

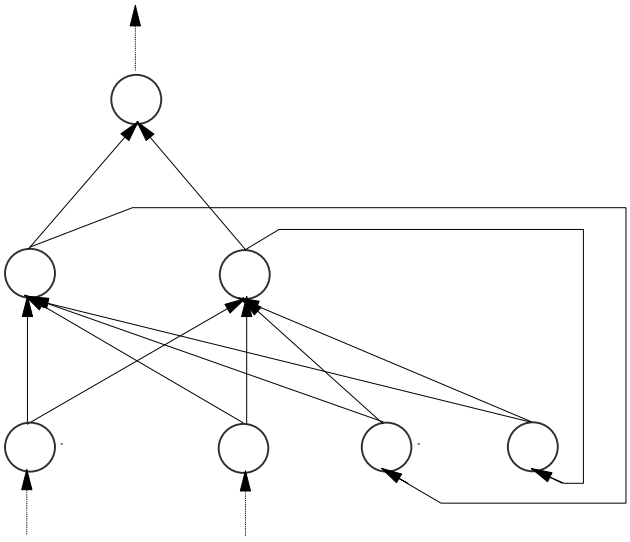


Figure 9 Simple recurrent network.

2.2.1.3 Real Time Recurrent Networks

Real Time Recurrent Networks (RTRN) [Williams, 1989] add temporal processing to the network by combining hidden and output units into a single processing layer and feeding back the output of the units in this layer to a concatenated input-output layer. Figure 10 is an example of a real time recurrent network.

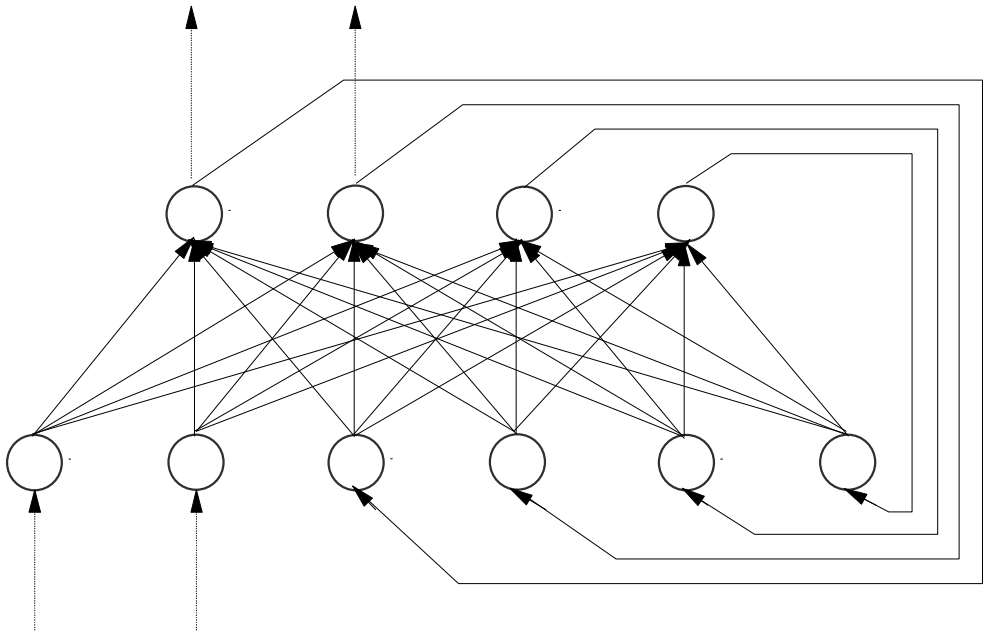


Figure 10 Real time recurrent network.

2.2.2 Network Training

Many learning algorithms have been developed for fixed architecture ANNs; both feed-forward and recurrent. The goal of any learning algorithm is to enable the ANN to adjust the weights and biases associated with the neurons which make up the network so that an input presented at the input units will cause the correct output to be activated at the output units. The learning algorithms can be divided into two broad groups: supervised learning algorithms and unsupervised learning algorithms.

In supervised learning algorithms, data is presented at the input units, and the output of the network is compared to an expected output pattern. An error is calculated based on the difference between the actual and expected output patterns, and that error is used to adjust the weights and biases of the network. This supervised learning is iterated until the network is considered to have been trained sufficiently.

Unsupervised learning algorithms teach the ANNs to respond to patterns or features in the input data. This technique is often used for data clustering or classification systems.

A learning rule commonly associated with multilayer feed-forward networks is the back-propagation rule [Werbos, 1974 and Rumelhart, 1986]. As discussed in the previous section, variations to the back-propagation rule exist for recurrent networks.

2.2.2.1 Error Back-Propagation

Also known as the *generalised delta rule*, the back-propagation rule is an example of supervised learning. Since the goal of the network is to produce the correct or expected output at the output layer given some input presented at the input layer, the goal of the training operation must be to reduce the difference between the expected output and the actual output. In order to do this an error function is defined which is some representation of the difference between the expected output and the network's actual output; then the training operation is an attempt to reduce the value of that error function.

In order to train a network a defined set of training examples are presented to the network, usually many times. As the training data set is presented, the value of the error function for the network is iteratively reduced by adjusting the weights of the network by some amount determined by the value of the error function.

One complete presentation of the training data set in the training operation is known as an *epoch*. Error back-propagation learning may be conducted in either *pattern* mode or *batch* mode. In pattern mode adjustment of the network weights is performed after the presentation of each training pattern, whereas in batch mode the network weights are adjusted after an entire epoch.

Error back-propagation involves two phases of computation. The first phase, known as the forward pass, involves the calculation of the activation of each neuron. The activation signals proceed forward from the input layer, through the network layer by layer until the output is calculated at the output layer. During this forward pass the network weights remain unchanged. The second phase, known as the backward pass, involves the error calculation and weight adjustment for each neuron. This backward pass begins at the output layer and propagates backward through the network layer by layer. The size of the weight adjustment for any neuron is governed by the value of the error function and a determination of the neuron's responsibility for that error value.

2.2.2.2 Generalisation and Over-fitting

A benefit offered by neural networks is their ability to generalise. Typically the goal of using a neural network is to train the network with known data in order to have the network predict the outcome in cases for which the outcome is not currently known. A serious problem for training neural networks for generalisation is the phenomenon of over-fitting. Over-fitting is the tendency for a neural network to learn the subtleties of the training data so well that it achieves an extremely low value for the error function during training, but is incapable of predicting unseen data with any degree of accuracy. This tendency to concentrate on the subtleties of the training data can cause the network to ignore other features of the data which

would otherwise allow good generalisation [Geman, 1992]. A technique known as stopped training, or early stopping, is commonly used in an attempt to overcome the problem of over-fitting [Sarle, 1995].

Several methods exist for estimating the generalisation error more accurately (these techniques however are not generally used in early stopping methods). Two of these methods are *k-fold* cross-validation and *leave-one-out* cross-validation.

In k-fold cross-validation, the data is divided into k equal sized subsets. Training is conducted on all but one of the subsets of data; the remaining subset being used as the validation set. The network is trained k times, each time using a different subset of data as the validation set. Leave-one-out cross-validation is just k-fold cross-validation with k equal to the number of training examples in the training set.

2.2.2.3 Stopping Training

Since over-fitting is a problem when training neural networks, it is important to determine the correct point at which to stop training. Probably the most commonly used technique is that known as early stopping, or stopped training [Sarle, 1995].

The early stopping method employs *split-sample* validation and involves training the network to the point that it has learned the training data sufficiently well for it to be capable of accurately predicting outcomes for previously unseen data. Available data is typically divided into two groups: the training data and the test data, with the test data being used after the network is trained to measure the ability of the network to generalise. In split-sample validation the training data is further divided into two groups, called the training set and the validation set, and during the training process the validation set is used to measure the quality of the network. Training of the network is stopped when the error for the validation set reaches a minimum. For any test using this method to be considered valid, no information about the test data or the performance of the network on the test data should be available during the training process.

2.2.2.4 Performance Measures

The goal of the back-propagation rule is to optimise the generalisation of the network, which it achieves by the minimisation of the network error. Typically, the error measured during training is the average sum-squared error [Rumelhart, 1986].

The sum-squared error is defined as:

$$E = \frac{1}{2}\sum_{p}\ \sum_{i}\left(t_{pi} - o_{pi}\right)^2$$

where the index ***p*** ranges over the training data set; the index ***i*** ranges over the set of output units; the variable t_{pi} indicates the target output for the ***i***th output unit for the ***p***th input pattern; and the variable o_{pi} indicates the actual output for the ***i***th unit for the ***p***th pattern.

The average sum-squared error is then calculated by dividing the sum-squared error by the total number of training patterns. In some circumstances it may be important to measure the error on a per-unit basis. The average *per-unit* sum-squared error can be calculated by dividing the average sum-squared error by the number of output units. One problem associated with the average per-unit sum-squared error is that it can be affected by the target variances of the output units.

The normalised error proposed by Pineda in [Pineda, 1988] addresses this problem. This error measure removes the effects of target variance, and ensures that the value of the error will lie between *0* and *1* for all networks. The normalised error proposed by Pineda is the sum-squared error divided by a quantity defined by Pineda as

$$E_{mean} = \frac{1}{2}\sum_{p}\sum_{i}\left(t_{pi} - \mu_i\right)^2$$

where the index ***p*** ranges over the training data set; the index ***i*** ranges over the set of output units; the variable t_{pi} indicates the target output for the ***i***th output unit for the ***p***th input pattern; and the variable μ_i is the average of the target values for the ***i***th output unit.

and the normalised error is given by

$$E_{norm} = \frac{E}{E_{mean}}$$

2.3 Previous Work

The various gradient descent techniques for finding a good set of weights in a fixed network suffer from several problems. The most notable of these are that they are computationally expensive and time consuming, and they have a tendency to become stuck in local minima. While several methods have been developed to overcome the problem of becoming stuck in local minima, these methods do not apply to all situations.

As a result of the problems associated with the gradient descent techniques, many researchers began to look to alternative methods for weight determination in fixed networks. Genetic algorithms, because of the global nature of their search, presented a promising avenue of research.

There are many different ways in which genetic algorithms and neural networks can be combined. Probably the most obvious is to represent the network weights in full precision on the chromosome, and use the genetic algorithm to optimise those weights. Obviously this method involves no learning or optimisation of the architecture of the network, the down side of which is that the genetic algorithm may be trying to fit weights to a network architecture which is in itself sub-optimal. The other end of the spectrum is to encode the complete network description in the chromosome and have the genetic algorithm evolve all facets of the network: the architecture, weights and biases. Work using both these approaches will be discussed, as well as some that falls between these two extremes.

2.3.1 Evolving Weights in Fixed Networks

A major obstacle with using genetic algorithms to evolve the weights of a fixed network is the encoding of the weights onto the chromosome. The weights of a neural network are generally real-valued and unbounded, whereas a chromosome in a genetic algorithm is generally a string of bits of some arbitrary length. Encoding real values onto such a chromosome presents problems both in the precision of the representation and the resultant length of the chromosome. The length of the chromosome impacts upon the size of the search space of the genetic algorithm, and the efficiency of the search. Obviously for very large neural networks with many connections, a chromosome encoded with full precision real numbers would be extremely long and difficult to deal with computationally.

Nicol Schraudolph and Richard Belew, in [Schraudolph, 1990], describe a mechanism for avoiding the sacrifice of either precision or search efficiency when encoding real-valued weights onto a fixed length chromosome. This technique, called *Dynamic Parameter Encoding* by Schraudolph and Belew, adapts the encoding scheme used for the chromosome such that the genetic algorithm concentrates its efforts on the most significant part of the weights early in the search, and on refining the search in the later stages.

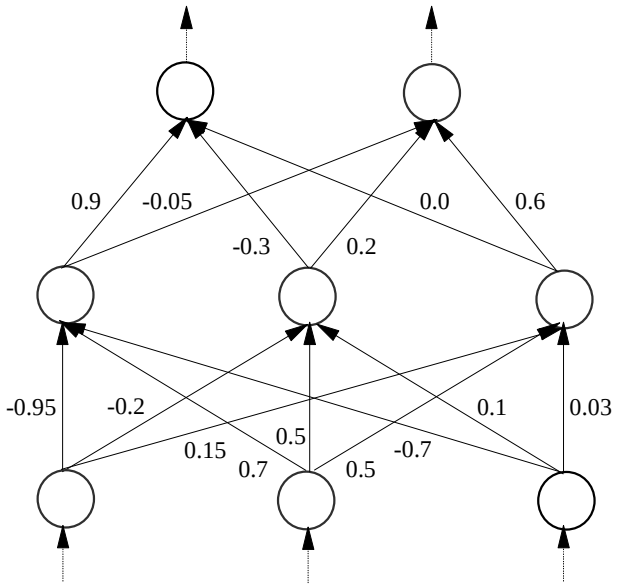
The assumption of this technique is that by using a coarse representation of the weights initially, the genetic algorithm will converge to a solution which, while not being the best solution, is a good solution. The search for the best solution may then continue, but since only the region already identified needs to be searched, the weights may now be re-encoded onto

the same chromosome with greater precision. This dynamic re-encoding of the search parameters and resultant refining of the search is the basis of the *Dynamic Parameter Encoding* technique.

Schraudolph and Belew present results which show that Dynamic Parameter Encoding can be a useful extension of the standard genetic algorithm search procedure.

[Montana, 1989] is an example of using real number coding instead of binary value coding for the chromosome. This avoids the problems associated with encoding real numbers onto the genetic algorithm's chromosome.

In the method described by Montana and Davis the genes in the chromosome are not bits, but real numbers. The chromosome is just an ordered list of real numbers representing the weights of the network. An example is shown in figure 11. Not shown in figure 11 is Montana and Davis' implementation of thresholding by the use of a threshold unit with inputs to each non-input unit in the network.



Chromosome: (0.9 -0.05 -0.3 0.2 0.0 0.6 -0.95 -0.2 0.15 0.7 0.5 0.5 -0.7 0.1 0.03)

Figure 11 Example of Montana and Davis' encoding of real-valued network weights.

The genes, or weights, are initialised to randomly selected real numbers between -1.0 and $+1.0$. With the exception of mutation, the values of the weights are never changed: the crossover operator changes only the position of the genes on the chromosome.

The crossover operator is implemented such that two parents create a single offspring in the following manner: for each of the offspring's non-input units, one parent is randomly selected and the incoming weights from the corresponding parent's unit are copied to the offspring.

The mutation operator adds a random number between -1.0 and $+1.0$ to the incoming weights of a selected number of non-input units.

This technique avoids the problems associated with encoding real numbers onto fixed length bit strings, but since the only way in which the weights can be changed is by mutation, it relies

to some degree on the best weights being part of the initial population. What is really evolved by this technique is the best combination of weights rather than the weights themselves.

Considered by several researchers to be a major obstacle to the effective use of genetic algorithms in training neural networks (and finite state machines, etc.) is the problem known variously as the *Multiple Symmetric Representations Problem* [Whitley, 1991], the *Permutations Problem* [Radcliffe, 1990], or the *Competing Conventions Problem.* Thisdescribes a situation which stems from the fact that in a fully connected network, hidden units differ only in their labelling. That is, in such a network, any two hidden units, along with their inputs and outputs, can be swapped without affecting the operation of the network.

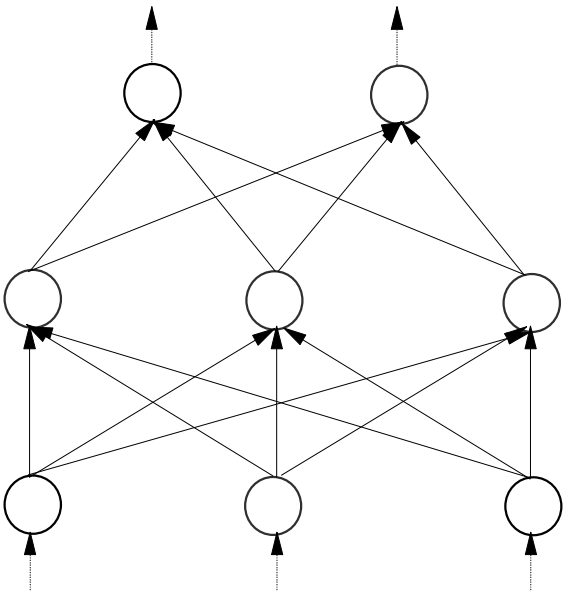


Figure 12 Fully connected feed-forward neural network.

Figure 12 illustrates a fully connected feed-forward neural network. It can be seen from this network that any hidden unit can be swapped with any other hidden unit without affecting the operation of the network. This means that there are in fact $3!$ representations of this neural network. If the representation of the network on the chromosome distinguishes units only by their label, then the genetic algorithm's search space is enlarged by $3!$ times, with the search becoming commensurably more difficult. This also affects the diversity of the population upon which genetic algorithms rely so heavily; while the population is numerically larger, the number of functionally unique networks is not.

There have been various solutions proposed to the competing conventions problem. Belew et al., in [Belew, 1990], develop a method of using a genetic algorithm to find a region of good fitness and set the network weights to good initial values, then have the search completed by back-propagation or some other gradient descent technique. This technique stems from the notion that due to the global nature of their search, genetic algorithms can readily find regions of good fitness, but are not as good as gradient descent techniques for reliably finding local minima. Belew et al. present findings which show that genetic algorithms can be effectively used to find initial weights advantageous for gradient descent techniques.

Montana and Davis, in [Montana, 1989], demonstrated that by implementing domain-specific genetic operators, and attempting to identify the functionality of hidden units during

recombination, the effects of the competing conventions problem could be minimised and performance improved.

In [Radcliffe, 1991], a solution was suggested which involves searching through the hidden units during recombination, identifying units which are identical and therefore would produce symmetric solutions, and using the information gained to guide the crossover operation.

In a similar vein, Amaral et al., in [Amaral, 1995], discuss reducing the search space by searching for and removing symmetric solutions.

In [Korning, 1995], Peter Korning suggests that using a fitness function more suited to the qualitative nature of genetic search than the least mean square fitness function traditionally used in back-propagation, may overcome the competing conventions problem.

2.3.2 Evolving Network Architectures

Balakrishnan and Honavar, in [Balakrishnan, 1995], present a good overview and classification of the current research in the area of evolutionary design of neural network architectures. Balakrishnan and Honavar also present a detailed bibliography of publications on this subject.

In [Miller, 1989], Miller, Todd and Hegde present their work on evolving the topology of feed-forward neural networks with a fixed number of units. In this work the neural network is represented by an $N \times (N+1)$ matrix, with the intersection of the rows (to units) and columns (from units and biases) indicating if a trainable connection existed between those units. Since each row of the matrix represents all the incoming connections to a given unit, the rows were considered to represent functional building blocks of the network. The crossover operator implemented by Miller et al. randomly selects a row index and swaps the corresponding rows of the parents to create two offspring. Miller et al. report some success with this method, albeit on fairly simple problems.

Brad Fullmer and Risto Miikkulainen describe an interesting mechanism for genetic encoding of neural networks based on the marker structure of biological DNA in [Fullmer, 1991]. The technique described by Fullmer and Miikkulainen allows the network architecture, weights and biases to be evolved through the use of the genetic algorithm.

In Fullmer and Miikkulainen's marker-based encoding technique the chromosome is represented by a list of integers. Each integer on the chromosome is interpreted as a start marker, an end marker, or a unit attribute depending upon the location of the integer on the chromosome. An integer x is interpreted as a start marker if $x \bmod 15$ is equal to 1, and as an end marker if $x \bmod 15$ is equal to 2. Start markers located in a unit definition are treated as an attribute of the unit. The chromosome is treated as a circular list, with unit attributes represented by integers located between start and end markers. The start marker, end marker, and unit attributes make up a unit definition. Integers located between end and start markers are not interpreted as part of the network. Unit attributes record the input and output specifications of the unit. Since start and end markers may change with evaluation of the new chromosomes, this representation of the units on the chromosome allows for evolution of the number of units, the connectivity between units, and the weights and biases of each unit.

Figure 13 shows an example of the mapping between a marker-based chromosome and a neural network. Each unit definition consists of a *start marker*, the *unit key*, the *unit's bias*, followed by pairs of connection weights encoded as the *target unit* and *weight* value. The definition for unit C has wrapped, demonstrating the circular nature of the chromosome. Crossover is implemented as standard two-point crossover, with the crossover points being randomly chosen anywhere on the chromosome. Three types of mutation are implemented depending upon where on the chromosome mutation is to occur.

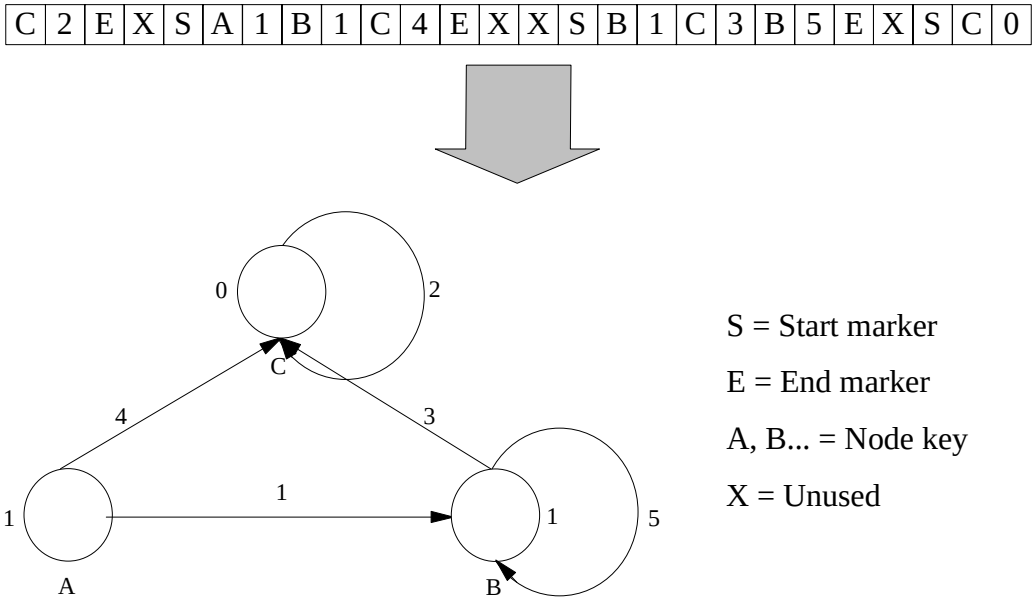


Figure 13 Example of marker-based chromosome to neural network mapping.

Mitchell Potter [Potter, 1992] describes an approach which applies a traditional genetic algorithm within the cascade-correlation architecture [Fahlman, 1990] in place of the gradient descent techniques normally applied. Potter presents results which indicate that the genetic cascade-correlation technique compares favourably with the standard quickprop learning method [Fahlman, 1988] in a domain where gradient information is available.

David Moriarty and Risto Miikkulainen present an extension to the marker-based encoding technique in [Moriarty, 1993] in which strategies for the game of Othello are evolved. Results obtained by this technique are quite promising.

In [Fogel, 1966], Fogel, Owens and Walsh introduced the notion of Evolutionary Programming. In that work, automata were evolved to predict symbol strings generated from Markov processes. In evolutionary programming the only method of variation is mutation: reproduction is asexual. Angeline, Saunders and Pollack use evolutionary programming techniques rather than genetic algorithms to evolve recurrent neural networks in [Angeline, 1994].

In [Yao, 1997], Xin Yao and Yong Liu describe a method of evolving networks using evolutionary programming techniques. The method described by Yao and Liu focuses on evolving network behaviours. The technique evolves the network architecture by the use of evolutionary programming, and uses simulated annealing to train the networks. Evolutionary programming techniques were adopted in an attempt to overcome the competing conventions problem. The technique described by Yao and Liu, known as EPNet, would seem to be reasonably complex and time consuming in terms of the evolution and training of networks,

and so most useful in applications for which network development and training is not time critical. Yao and Liu claim that their technique results in compact networks which show good generalisation ability.

Along the same lines of separating the evolution of the network from the training of the weights, Esparcia-Alcázar and Sharman [Esparcia-Alcázar, 1997] present a technique for the evolution of recurrent neural networks which combines genetic programming [Koza, 1992] and simulated annealing. Genetic programming is an extension of the genetic model into the space of programs in some functional language. That is, the structures undergoing adaptation are not fixed-length strings of bits in which the potential solution is encoded, but rather are parse trees that represent programs which, when executed, are the candidate solutions. In this work, Esparcia-Alcázar and Sharman express recurrent neural networks as a set of expression trees and use genetic programming to evolve these expression trees. Esparcia-Alcázar and Sharman conclude that although their technique shows promise, because it is computationally expensive it is not yet suited for real time applications.

Syed, in [Syed, 1995], investigates modifying genetic algorithm parameters to both find network weights and evolve the network architecture for recurrent neural networks. Syed reports some success with variations in the genetic algorithm such as gray scale encoding, mixed population size, and tournament selection. Syed uses a neural network architecture similar to that of the Hopfield network in his work. The standard genetic algorithm is used to evolve and train a recurrent network to solve the XOR problem. Variations of the genetic algorithm are then compared to the results obtained with the standard algorithm, with the most successful variations being used to evolve networks capable of solving more difficult problems.

In [Moriarty, 1996a], Moriarty and Miikkulainen describe a novel variation of applying evolutionary algorithms to evolving neural networks which they call SANE (Symbiotic, Adaptive Neuro-Evolution). With the SANE method, the individuals in the population are not complete networks but individual neurons. The SANE method endeavours to evolve individual neurons which are then combined to form a complete network. Each individual neuron's fitness is a function of the fitness of each of the networks in which the neuron is a participant. The assumption is that since no individual neuron can perform well enough alone to solve whatever problem the network is being applied to, the neurons being evolved must form a symbiotic relationship with other neurons in order to produce a complete network which performs well. Since no single neuron can provide the solution, individuals must rely on other, different individuals with which it combines in order to achieve a high fitness value. Because no individual can by itself produce a network of sufficient fitness, the population of neurons stays inherently diverse, thus allowing the genetic algorithm to search a wider range of solutions in the solution space.

Moriarty and Miikkulainen further refine SANE in [Moriarty, 1996b], in which the notion of network blueprints is introduced. Network blueprints are described as a mechanism for specifying a collection of neurons which have performed well together. SANE maintains a population of blueprints, and uses genetic selection and recombination to evolve a layer of such blueprints at a level above the evolution of individual neurons. The assumption is that by evolving these network blueprints, the genetic algorithm is able to exploit the best performing networks found during evolution. In this way, the genetic search at the neuron level is able to evolve individual neurons well suited to specific tasks which are then combined into networks. The second level genetic search is then able to search these collections of neurons for highly fit networks. Better performing neurons will tend to

participate in a greater number of networks, so the search will be biased towards the better performing neurons.

Moriarty, in [Moriarty, 1997] presents the SANE method and its application to sequential decision tasks. Moriarty concludes that the SANE method out-performs the current methods for learning decision strategies in complex problems.

Chapter 3
2DELTA-GANN

3.1 Overview

2DELTA-GANN is a technique developed by Rajendra Krishnan and described in [Krishnan, 1994a] and [Krishnan, 1994b] which uses genetic algorithms to train fixed architecture feed-forward neural networks. The approach used in 2DELTA-GANN is to have the genetic algorithm evolve some change, or delta value, to the weights and biases of the neural network being trained. This is done by modifying the weights and biases of the network by some value according to some combination of fixed rules. Rather than have the genetic algorithm evolve the actual weights and biases of the network, only the way in which the rules are to be applied and the delta values are evolved. As a result, the chromosome being modified by the genetic algorithm does not need to represent each weight and bias of the network; only the method by which the rules are to be applied and the delta values need be represented.

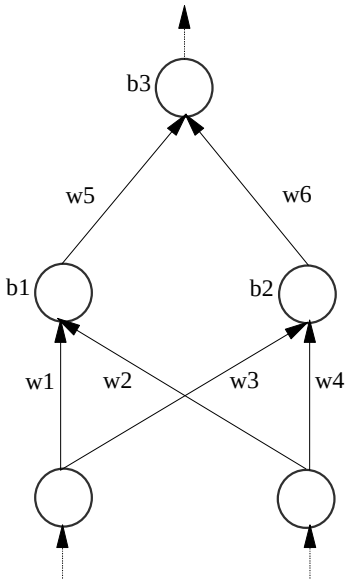
The gene structure used by Krishnan is an attempt to overcome the problems associated with encoding real numbers onto a chromosome represented as a bit string. In the 2DELTA-GANN method, each gene on the chromosome is a composite structure representing either a weight or a bias from the neural network. The gene is composed of three rule bits and two floating point values. The floating point values are manipulated in accordance with the rule bits to apply a change to the weights and biases of the network. The three rule bits are denoted $x1$, $x2$ and $x3$; and the two floating point values are denoted $delta1$ and $delta2$.

The basic concept of the 2DELTA-GANN method is to use the rule bits to specify a simple heuristic to apply to the $delta2$ value, which is then used to modify the value of $delta1$. Finally, $delta1$ is used to modify the network weight or bias associated with the gene.

Modified methods of crossover and mutation are used to manipulate the chromosome formed from the combination of the gene structures. Because of the modifications to the crossover and mutation operators, the representation of the genetic structure can be somewhat simplified. In this method, each gene is represented as three bits, corresponding directly to the rule bits $x1$, $x2$ and $x3$. Each gene then has associated with it two floating point values, corresponding to $delta1$ and $delta2$. In reality, only the rule bits are combined to form the chromosome. This allows the chromosome to be kept to a minimum length, and further means that the floating point values do not need to be represented as bit strings on the chromosome. The values of $delta1$ and $delta2$ become attachments to the chromosome. The modified crossover and mutation operators are aware of the chromosome structure and the floating point values associated with each gene. These modified operators are designed such that if any rule bits of a particular gene are affected by the operation, then the $delta1$ and $delta2$ values associated with that gene are also affected in some meaningful way.

2DELTA-GANN utilises the GA$_{UCSD}$ genetic algorithm package from [Schraudolph, 1992] which allows a user supplied fitness function. The standard crossover and mutation functions of GA$_{UCSD}$ were modified for the new gene structure.

Figure 14 shows a simple XOR network, a sample chromosome, and possible corresponding ***delta1*** and ***delta2*** values for each gene on the chromosome. The chromosome is applied to the neural network by first applying the heuristic specified by the rule bits ***x1***, ***x2*** and ***x3*** for each gene to the ***delta1*** and ***delta2*** values corresponding the those genes. The weight or bias associated with each gene is then modified by the value of ***delta1***.



w1	w2	w3	w4	w5	w6	b1	b2	b3	
010	011	111	011	101	010	010	010	111	x1x2x3
0.3	-0.7	0.5	-0.2	0.4	-0.6	-0.4	-0.3	0.2	delta1
0.02	-0.1	0.09	0.04	-0.1	0.15	-0.06	0.2	-0.03	delta2

Figure 14 The XOR network with a sample chromosome.
Adapted from 2DELTA-GANN: A New Method of Training Neural Networks Using Genetic Algorithms.

The chromosome is applied to the network by interpreting the rule bits in the following way:

```
if x1 = 1 then
        if x2 = 1 and x3 = 1 then
                double delta2
        else
                halve delta2
        endif
        add delta2 to delta1
endif
add delta1 to weight (or bias)
```

In the original work with 2DELTA-GANN, two different types of genetic algorithm were used: a generational replacement algorithm, and a steady state algorithm. The neural networks considered by Krishnan were all feed-forward networks.

2DELTA-GANN implements a modified form of uniform crossover. Since the gene structure implemented by 2DELTA-GANN involves a composite structure, the chromosome is not a simple bit string, so the crossover operator was modified to accommodate the composite gene structure. As part of the uniform crossover operation described above, the 2DELTA-GANN crossover operator will also crossover the delta values of any genes whose rule bits were affected by the uniform crossover.

Similarly, the mutation operator was also modified to accommodate the composite gene structure. Each bit on the chromosome is a candidate for mutation with some arbitrary probability. If a gene has any rule bit mutated, then the ***delta****1* and ***delta****2* values for that gene are assigned randomly generated values.

3.2 Analysis

3.2.1 Implementation

During inspection of the 2DELTA-GANN source code in preparation for making the changes necessary to support recurrent networks, a minor coding error which had the effect of causing no crossover of the rule bits to take place was noticed. Crossover of the delta values would still occur as though crossover of the rule bits had taken place. This meant that the only way the rule bits would ever change was by mutation.

Further analysis of the 2DELTA-GANN code revealed what is considered to be a flaw in the implementation. The algorithm as implemented by Krishnan is:

Step 1: Initialise the network weights to random values
Step 2: Create a population with randomly assigned rule bits and delta values
Step 3: Evaluate each member of the population
Step 4: If the best performer is an improvement on the previous best, then apply the delta values of the best performer to the network weights to calculate new network weights
Step 5: Rank and select members of the population for reproduction
Step 6: Perform crossover
Step 7: Perform mutation
Step 8: If not finished goto Step 3

The aim of the algorithm is to find the best set of weights for the network, and this is achieved by modifying the weights at each generation (by application of the delta values) and retaining the single best set of weights for use with the following generation.

While this at first glance looks valid, the problem is at Step 4. The performance of the population is evaluated at Step 3, and the ranking and selection at Step 5 is based upon that performance evaluation, but the weights to which the delta values will be applied are possibly changed at Step 4. This means that the ranking and selection at Step 5 may not have selected the fittest members of the population for reproduction *in relation to the new set of weights*.

The following algorithm is believed to be correct:

Step 1: Initialise the network weights to random values
Step 2: Create a population with randomly assigned rule bits and delta values
Step 3: Evaluate each member of the population
Step 4: If the best performer is an improvement on the previous best, then
  Step 4.1: Apply the delta values of the best performer to the network weights to calculate new network weights
  Step 4.2: Re-evaluate each member of the population against the new network weights
Step 5: Rank and select members of the population for reproduction
Step 6: Perform crossover
Step 7: Perform mutation
Step 8: If not finished goto Step 3

By adding the re-evaluation to Step 4, the ranking and selection which takes place at Step 5 will be based upon the performance of the members of the population in relation to the new weights. This is believed to be a more correct implementation of the theory than Krishnan's original implementation. Later in this document comparative results from tests conducted using both the original 2DELTA-GANN code and modified versions of the code which implement the changes discussed above will be presented.

The presence of the coding error and the error in implementation are surprising given the promising results described by Krishnan. This led to a closer consideration of the results which had been achieved and the implication of those in conjunction with the errors in implementation. 2DELTA-GANN is intended to achieve its goal of finding a set of weights for the network by evolving a set of rules to be applied to the delta values for each weight or bias of the network. The basic construction of the algorithm and the nature of the genetic search employed means that as the search progresses, not only do the weights and biases of the network change, but the delta values for each unit are modified both by recombination and mutation. This means that by the use of this algorithm it is intended that a stable set of rules will be evolved to be applied to a constantly changing set of delta values. Looking again at the algorithm for updating the delta values and network weights:

```
if rule bit 1 = 1 then
        if rule bit 2 = 1 and rule bit 3 = 1 then
                double delta2
        else
                halve delta2
        endif
        add delta2 to delta1
endif
add delta1 to weight (or bias)
```

It can be seen from this algorithm that the weight (or bias) value will *always* be updated by the ***delta****1* value, and what is being discovered or learned by the algorithm is the *rate of change* of the ***delta****2* value (and since ***delta****1* may be modified by ***delta****2*, the rate of change of ***delta****1*), and whether the ***delta****1* value should be modified by ***delta****2*.

It can also be seen that ***delta****2* will never change sign. That is, a ***delta****2* which is initially negative will remain negative, and an initially positive ***delta****2* will remain positive. This

means that if the delta values for a particular weight or bias (corresponding to a unit) are such that they cause the weight or bias to move in the wrong direction, the only way this can be overcome is by crossover or mutation of the delta values, never by crossover or mutation of the rule bits.

It is contended that because the values of ***delta****1* and ***delta****2* are changing almost continuously (and possibly exponentially), and the weights and biases of the network are also changing, the genetic algorithm is not capable of evolving a stable set of rules for the application of the delta values to the weights and biases of the network, even though what is intended is that the rate of change of the delta values be evolved. This work contends that any rules being evolved by the genetic algorithm are of little or no use in directing the search.

To test this hypothesis a variation of 2DELTA-GANN was constructed which uses the following algorithm to determine how to change the delta values:

```
if Rand() > 0.5
        if Rand() > 0.5 then
                double delta2
        else
                halve delta2
        endif
        add delta2 to delta1
endif
add delta1 to weight (or bias)
```

The change is only in the application of the delta values. The crossover and mutation operators for both the rule bits and the delta values are still in place and operational. By applying the deltas to the weights and biases of the network randomly, any advantage gained by the recombination of highly fit schemata for the rule bits is lost (since the rules as learned by the genetic algorithm are no longer used). Some benefits of the genetic algorithm remain; namely the advantage of a population based search, the advantages of recombination are still effective for the delta values themselves, and random mutation of the delta values. It is expected that the tests using this algorithm will be comparable with the tests using the heuristic rules implemented by 2DELTA-GANN.

In order to determine if the 2DELTA-GANN method is a viable method for training recurrent neural networks a new fitness function was developed. This involved removing the entire neural network code and writing new neural network evaluation code which could handle several different types of recurrent neural networks.

3.2.2 The Disruptive Effect of Crossover

The crossover method implemented by 2DELTA-GANN uses uniform crossover to determine from which parent the offspring inherit individual rule bits. Most of the original tests conducted by Krishnan on feed-forward networks seem to have used a probability of crossover for any bit position of $0.15 - 0.2$. As evidenced by the references following, there are differing opinions in the genetic algorithm community as to the relative benefits of the

different types of crossover (single point, two point, uniform etc.), and even as to the effects of each type of crossover.

In his thesis Krishnan refers to [Caruana, 1989] when he notes that *"Uniform crossover with probability of crossover* $\boldsymbol{p} = 0.5$ *is highly disruptive - it has high distributional bias (50%) but low positional bias. As p is reduced the distributional bias is reduced"* [Krishnan, 1994a, p14]. Krishnan also refers to [Spears, 1991] and notes that *"Studies by DeJong and Spears show that by using a probability of less that 0.1, for all except very short defining lengths, uniform crossover is actually less disruptive than both one and two point crossover"* [Krishnan, 1994a, p12].

According to Mitchell in [Mitchell, 1996], Spears and DeJong believe what they term as *parameterized uniform crossover* is superior to other forms of crossover. With this method, the probability $\boldsymbol{p}$ that crossover occurs for any bit position is typically given by $0.5 \leq \boldsymbol{p} \leq 0.8$.

Mitchell also notes that *"it is common in recent GA applications to use either two-point crossover or parameterized uniform crossover with* $\boldsymbol{p} \approx 0.7 - 0.8$ *"* [Mitchell, 1996, p173].

Since 2DELTA-GANN implements a gene structure which is not just a simple bit string, the schemata and building blocks referred to by Holland and Goldberg are not just strings of bits, but groups of real numbers. How recombination affects the modified gene structure implemented here, and just how disruptive the different types of crossover might be, is not well understood.

Chapter 4
Experiments and Analysis

4.1 Experimental Plan and Description of Methods

Following is the experimental plan adopted for this thesis:

- re-train the feed-forward networks tested by Krishnan in his thesis, both with the original 2DELTA-GANN code and with the modified versions of the algorithm.
- compare the results obtained for the feed-forward networks with the various forms of the algorithm to verify that the modifications improved performance.
- train the feed-forward networks using the back-propagation technique and compare the results with those achieved by the various forms of the 2DELTA-GANN algorithm.
- train a number recurrent networks with the modified versions of the algorithm to determine the performance of the technique with recurrent networks.
- train the recurrent networks using the back-propagation technique appropriate to the recurrent architecture, and compare the results with those achieved by the various forms of the 2DELTA-GANN algorithm.
- evaluate a method of early stopping with the 2DELTA-GANN technique to determine if the effects of over-training can be lessened.

The feed-forward and recurrent networks tested are described in the following sections. The variations of 2DELTA-GANN for which results are presented are described below in table 1.

Method	Description
Original	The original 2DELTA-GANN. The results presented are from the original results generated by Krishnan for his thesis.
Re-run	The original 2DELTA-GANN re-run with delta values selected to provide results for comparison with the new methods.
Repaired	The original 2DELTA-GANN code with the defect which prevents crossover of the rule bit from occurring repaired. *See page 25*.
Re-evaluation	The repaired version of the code with the re-evaluation step added. *See page 25*.
No Rules	The repaired version of the code with the re-evaluation step added and the random application of the delta values. *See page 26*.

Table 1 Variations to the 2DELTA-GANN method.

4.2 Network Attributes

The networks tested include networks whose inputs and outputs are binary, and networks whose input and outputs are continuous, or real-valued. All networks tested use either the linear activation function or the sigmoidal activation function. This presents a small problem for those networks whose outputs are required to be binary. Since the output of both the linear function and the sigmoid function are continuous, the output from a unit which is expected to be binary needs to be interpreted. Generally the sigmoidal activation function would be implemented for units whose output is to be interpreted as binary, and for training purposes an output from such a unit of ≤ 0.4 is interpreted as 0, and an output of ≥ 0.6 as 1. Between those values the unit's output is not defined. For networks with continuous outputs a unit is considered to be trained if its output is within a given percentage of the target output, or is within a fixed tolerance. A value of 10% was used for all continuous output networks, except for the digital to analogue converter network for which a fixed tolerance of 0.03 was used.

Krishnan noted in his thesis that the 2DELTA-GANN method is sensitive to the bounds of the initial delta values, but did not document the values used in his experiments. New results obtained by re-running the original code with delta bound values which are either a reasonable estimate of the values used by Krishnan, or those which after a short manual search were found to produce reasonable results, are presented here for comparison.

The delta values used for each network are unchanged for the different methods except where indicated.

4.3 Performance Measures

Two different measures of performance are used and reported. For the tests which are compared with the original 2DELTA-GANN code and results, a variant of the sum-squared error is used. This is to maintain consistency with the results produced by the original 2DELTA-GANN and to allow direct comparison with those results. The sum-squared error is defined as:

$$\boldsymbol{E} = \frac{1}{2}\sum_{\boldsymbol{p}} \sum_{\boldsymbol{i}} \left(\boldsymbol{t}_{\boldsymbol{pi}} - \boldsymbol{o}_{\boldsymbol{pi}}\right)^2$$

where the index $\boldsymbol{p}$ ranges over the training data set; the index $\boldsymbol{i}$ ranges over the set of output units; the variable $\boldsymbol{t}_{\boldsymbol{pi}}$ indicates the target output for the $\boldsymbol{i}$th output unit for the $\boldsymbol{p}$th input pattern; and the variable $\boldsymbol{o}_{\boldsymbol{pi}}$ indicates the actual output for the $\boldsymbol{i}$th unit for the $\boldsymbol{p}$th pattern.

The original 2DELTA-GANN results reported the error as $2 \times \boldsymbol{E}$, so for consistency the same is done here when comparing results with the original 2DELTA-GANN.

For the tests performed on the recurrent networks (other than the sequence generator), the performance measure is the normalised error proposed by Pineda as described earlier in section 2.2.2.4.

4.4 Re-analysis of 2DELTA-GANN Results

The variations to 2DELTA-GANN described in table 1 are tested on the XOR network, the 4-2-4 encoder network, and the digital to analogue converter network as described by Krishnan in his thesis, and the results compared with the results obtained by him. Each method is tested for fifty independent runs for each of the three networks.

Each of the networks is also trained using the back-propagation technique (using the program supplied with [McClelland, 1988]), and these results are compared with results obtained using the various 2DELTA-GANN techniques. For the purpose of comparison with back-propagation, a single evaluation of an individual of the genetic algorithm's population of networks is considered to be computationally equivalent to a single (forward or backward) pass of the back-propagation technique. A generation requires evaluation of each member of the population for the entire training data set, so in terms of back-propagation a generation is typically computationally equivalent to a number of epochs. The number of epochs a generation is equivalent to varies with the size of the population and whether the method being tested includes the re-evaluation step.

For example, for the original 2DELTA-GANN method and a population of 200 networks which takes 15 generations to train, each network will have been evaluated 15 times for any one input pattern, so there will have been $15 \times 200 = 3000$ network evaluations per input pattern. For the methods which introduce the re-evaluation however, each network will have been evaluated twice for each generation, so there will have been $15 \times 2 \times 200 = 6000$ network evaluations per input pattern.

Similarly, for back-propagation, a training period of 175 epochs will have resulted in $175 \times 2 = 350$ network evaluations per input pattern (this takes into consideration the forward and backward passes of the back-propagation technique).

For ease of comparison, the number of epochs for the back-propagation techniques and the number of trials for the various forms of 2DELTA-GANN will be converted to, and quoted as, the number of network evaluations per input pattern.

Test results are presented for:

- the number of successful runs.
- the lowest error $(2 \times \boldsymbol{E})$ of any run.
- the average error $(2 \times \boldsymbol{E})$ over the fifty runs.
- the fewest generations for any run.
- the average number of generations for all runs.
- the fewest evaluations for any run.
- the average evaluations for all runs.

A good result for these tests would show the number of successful runs at a maximum, and all other measures at a minimum.

Conclusions drawn from the results presented for the feed-forward networks are described in following sections.

4.4.1 The XOR Network

The XOR network shown in figure 15 was selected by Krishnan as a good test network because the solution space is known to contain local minima, thus making the problem a little more difficult for gradient descent techniques. This network also demonstrates the competing conventions problem quite clearly (the hidden units are interchangeable without change to the performance of the network). The purpose of the network is to compute the exclusive-OR of the two inputs. The XOR network uses the sigmoidal activation function for all hidden and output units. Input to and output from the network are binary. Training of the network is stopped when, for all input patterns, for a target value of 0 the output is ≤ 0.4, and for a target value of 1 the output is ≥ 0.6; or after 8000 trials if this condition is not reached before then. The results for each of the methods described in table 1 are presented in table 2.

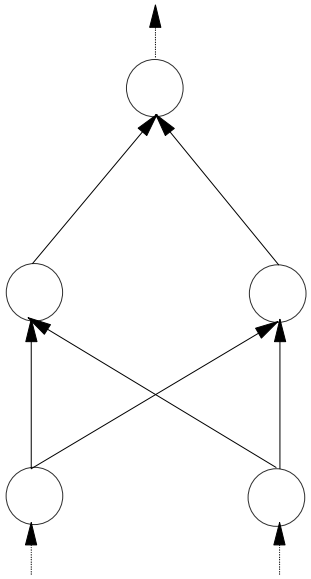


Figure 15 The XOR network.

XOR Network							
	Successful	$2 \times E$		**Generations**		**Evaluations**	
Method	**Runs**	**Lowest**	**Average**	**Fewest**	**Average**	**Fewest**	**Average**
Original	46	0.0000	0.0998	4	12.0	800	2392
Re-run	38	0.0442	0.3201	5	16.6	1000	2328
Repaired	32	0.0740	0.3383	5	19.8	1000	3952
Re-evaluation	45	0.0471	0.2814	5	13.5	1604	5096
Re-evaluation*	50	0.0000	0.0081	1	6.4	192	2342
No Rules	45	0.0146	0.3168	4	12.3	1398	4666
Back-Propagation	45	0.2789	0.3514	n/a	n/a	268	2168

Population = 200, Probability of Crossover = 0.2, Probability of Mutation = 0.005
***** Results achieved by varying the initial bounds for *delta1* and *delta2*.

Table 2 Results for the XOR network over 50 runs.

In isolation, these results would seem to indicate that repairing the crossover defect had a negative effect on the outcome of the search, but that adding the re-evaluation step improved performance again. The important outcome of these tests is the observation that replacing the

application of the rules to update the delta values with a random application has actually led to an increase in performance. While not definitive, these results support to a certain extent the hypothesis outlined earlier that any rules evolved by the algorithm are of little or no benefit to the performance of the search.

These results also indicate that the best results from the 2DELTA-GANN methods compare very favourably with the results obtained from the back-propagation technique, in terms of both classification accuracy and training time.

4.4.2 The 4-2-4 Encoder Network

The 4-2-4 encoder network shown in figure 16 was selected by Krishnan because it is a relatively easy problem which is readily scalable. This network also demonstrates the problem of competing conventions. The purpose of the network is to encode, then decode, a 4-digit value. The usefulness of the network is in its ability to compress and decompress data. In a real environment, data could be presented at the inputs and the output of the hidden layer stored as a compressed representation of the input data. At some later stage the data could be expanded by presenting the stored data to the output layer. The 4-2-4 encoder network uses the sigmoidal activation function for all hidden and output units. Input to and output from the network are binary. Training of the network is stopped when, for all input patterns, all units whose target value is 0 are ≤ 0.4, and all units whose target value is 1 are ≥ 0.6; or after 10000 trials if this condition is not reached before then. The results for each of the methods described in table 1 are presented in table 3.

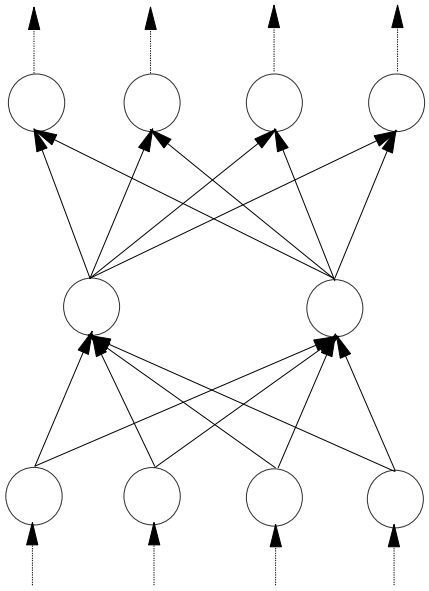


Figure 16 The 4-2-4 encoder network.

4-2-4 Encoder Network							
	Successful	$2 \times E$		Generations		Evaluations	
Method	Runs	Lowest	Average	Fewest	Average	Fewest	Average
Original	46	0.0987	0.3586	17	30.7	3400	6136
Re-run	47	0.0588	0.3447	10	17.8	2000	3564
Repaired	48	0.1067	0.3023	9	17.0	1800	3392
Re-evaluation	48	0.1772	0.5272	8	15.6	2994	5970
Re-evaluation*	50	0.0323	0.4825	11	19.9	4064	7642
No Rules	50	0.2770	0.5633	7	13.6	2494	5134
Back-Propagation	50	0.3704	0.5519	n/a	n/a	54	118

Population = 200, Probability of Crossover = 0.15, Probability of Mutation = 0.005
***** Results achieved by varying the initial bounds for *delta1* and *delta2*.

Table 3 Results for the 4-2-4 encoder network over 50 runs.

As with the results for the XOR network, repairing the crossover defect has caused some degradation of performance. Adding the re-evaluation step has decreased performance with regard to the error attained, but has achieved the same number of successful runs. Removal of the application of the rules has led to a further decrease in performance with regard to the error, but has improved the success rate of the method. These results give some further support to the earlier hypothesis that the evolution of the rules is of little benefit. Varying the bounds for the initial values of the deltas has allowed 2DELTA-GANN to achieve a very low error value.

These results show that while back-propagation in this case out-performed 2DELTA-GANN with regard to the training time, 2DELTA-GANN was able to attain a lower error value, indicating that 2DELTA-GANN is capable of producing results which in terms of classification accuracy are comparable to or better than those produced by back-propagation.

4.4.3 The Digital to Analogue Converter Network

The digital to analogue converter network shown in figure 17 was selected by Krishnan because it demonstrates the ability of the 2DELTA-GANN method to train a network capable of distinguishing precisely between outputs. This network is a test of the fine learning ability of the algorithm. The digital to analogue converter network uses the sigmoidal activation function for all hidden and output units. Input to the network is binary, and output from the network is continuous. Training of the network is stopped when, for all input patterns, the output produced at the output unit is within *0.03* of the target value; or after *50000* trials if this condition is not reached before then. The results for each of the methods described in table 1 are presented in table 4.

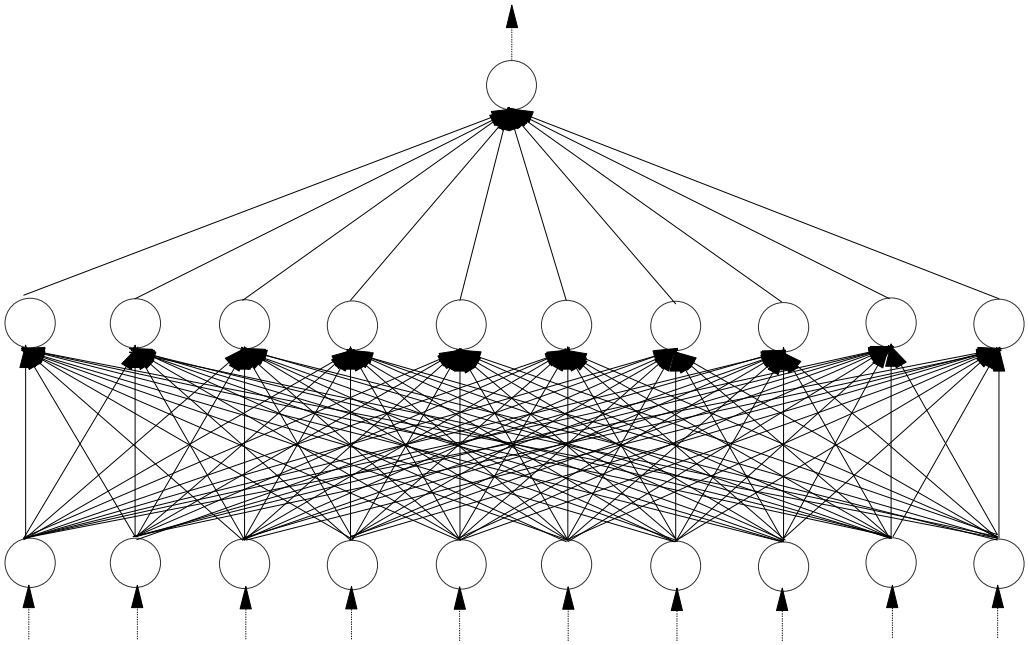


Figure 17 The digital to analogue converter network.

Digital to Analogue Converter Network							
	Successful	$2 \times E$		**Generations**		**Evaluations**	
Method	**Runs**	**Lowest**	**Average**	**Fewest**	**Average**	**Fewest**	**Average**
Original	48	0.0012	0.0025	13	23.0	13000	23040
Re-run	49	0.0011	0.0025	12	17.4	12000	17440
Repaired	49	0.0005	0.0026	11	16.5	11000	16520
Re-evaluation	50	0.0014	0.0026	13	17.2	25040	33222
No Rules	50	0.0016	0.0028	16	20.4	31834	39438
Back-Propagation	50	0.0009	0.0014	n/a	n/a	184	222

Population = 1000, Probability of Crossover = 0.2, Probability of Mutation = 0.005

Table 4 Results for the digital to analogue converter network over 50 runs.

These results show that while the repaired version of 2DELTA-GANN attained a very low error value, overall the different variations of 2DELTA-GANN are comparable. The important observation from these results is that the removal of the application of the rules gives performance comparable to the algorithm which implements the application of the rules. This again gives support to the hypothesis that the algorithm is not capable of learning rules which are of any real benefit to the search.

None of the 2DELTA-GANN methods were able to match the performance of back-propagation for this problem in terms of the training time. Back-propagation also produced more consistent results than the 2DELTA-GANN methods. This is to be expected for problems for which gradient information is readily available. In terms of classification accuracy, the results produced by the various forms of 2DELTA-GANN compare quite favourably with those produced by back-propagation.

4.4.4 Discussion of Results

The results presented in the previous sections for the feed-forward networks indicate that the modifications to the 2DELTA-GANN code and algorithm have improved the performance of the technique in some cases, at least with regard to training feed-forward networks. While in some cases the network error may have been slightly larger than the original method, the success rate of the modified algorithm is slightly better.

An important observation from these tests is that any rules actually evolved by the method are of little or no benefit to the overall performance of the technique. The demonstration that a random application of the delta values to the weights and biases of the network performs similarly to the application of the delta values by the evolved rules serves to show that the technique evolves no rules beneficial to the search.

The evaluation of the back-propagation technique and the comparison of the results achieved by that technique with those achieved by the various forms of the 2DELTA-GANN algorithm serve to show that 2DELTA-GANN compares favourably with back-propagation. While in some cases back-propagation out-performs 2DELTA-GANN in terms of training time, 2DELTA-GANN is shown to be capable of producing networks whose classification accuracy is comparable to those produced by back-propagation.

In his thesis [Krishnan, 1994a, p53], Krishnan presented a comparison of the results obtained by 2DELTA-GANN and results obtained with other methods of using genetic algorithms to train neural networks by Whitley et al. in [Whitley, 1989] and [Whitley, 1991]. Table 5 reproduces some of the data presented by Krishnan, as well results attained by the no rules method of 2DELTA-GANN for some of the networks described by Krishnan. These results indicate that 2DELTA-GANN compares quite favourably with this earlier work on using genetic algorithms to train neural networks.

A description of the 4-2-3 minimal adder and 4-4-3 adder networks is given in [Krishnan, 1994a]. For consistency, where applicable the number of trials presented by Krishnan has been converted to the number of evaluations.

Network		Whitley	2DELTA-GANN Original	2DELTA-GANN No Rules
XOR	Success Rate	100%	92%	90%
	Evaluations	500	2600	4666
	Population	100	200	200
4-2-4 Encoder	Success Rate	100%	92%	100%
	Evaluations	25000	6700	5134
	Population	1000	200	200
4-2-3 Minimal Adder	Success Rate	100%	52%	n/a
	Evaluations	100000	136300	n/a
	Population	3000	1000	n/a
4-4-3 Adder	Success Rate	0%	52%	n/a
	Evaluations	1250000	117300	n/a
	Population	2000	1000	n/a

Table 5 Comparison of 2DELTA-GANN with an earlier GA-NN approach.

4.5 Recurrent Networks

The variations of 2DELTA-GANN described in table 1 used to train recurrent networks are the *Re-evaluation* method and the *No Rules* method. These methods are tested on several recurrent networks to determine their ability to train different recurrent architectures.

Back-propagation techniques are also compared with the results achieved by 2DELTA-GANN for some of the recurrent networks. For the sine network (described in the following sections) recurrent back-propagation is trialed and the results presented. For the sunspot networks (also described later), published data for recurrent back-propagation and real time recurrent learning is presented. As with the feed-forward networks, epochs for the back-propagation methods will be converted to, and quoted as, evaluations.

Conclusions drawn from the results presented for the recurrent networks are described in following sections.

4.5.1 The Sequence Generator Network

The sequence generator network shown in figure 18 is described as a modified Jordan network by McClelland and Rumelhart in [McClelland, 1988]. The network is similar to a real time recurrent network in that the recurrent links are taken from the output layer to a context layer. The distinguishing features of this network are that it has an extra hidden layer (which takes input from both the input layer and the context layer), and the self-recurrent links on the context units. For this test the weight of each self-recurrent link was fixed at 0.5.

The sequence generator network uses the sigmoidal activation function for all hidden and output units. Input to and output from the network are binary. Training of the network is stopped when, for all input patterns, all units whose target value is 0 are ≤ 0.4 and all units whose target value is 1 are ≥ 0.6; or after 100000 trials if this condition is not reached before then.

The purpose of the network is sequence generation. Properly trained, the network will turn on the output units in sequence, either left to right if the pattern over the input units is (1010), or right to left if the pattern over the input units is (0101).

Since the back-propagation code from [McClelland, 1988] used in 2DELTA-GANN as the network evaluation code (the fitness function for the genetic algorithm) is capable of training and testing this type of recurrent network, results are presented from tests run with the unmodified 2DELTA-GANN code for comparison with the modified algorithms.

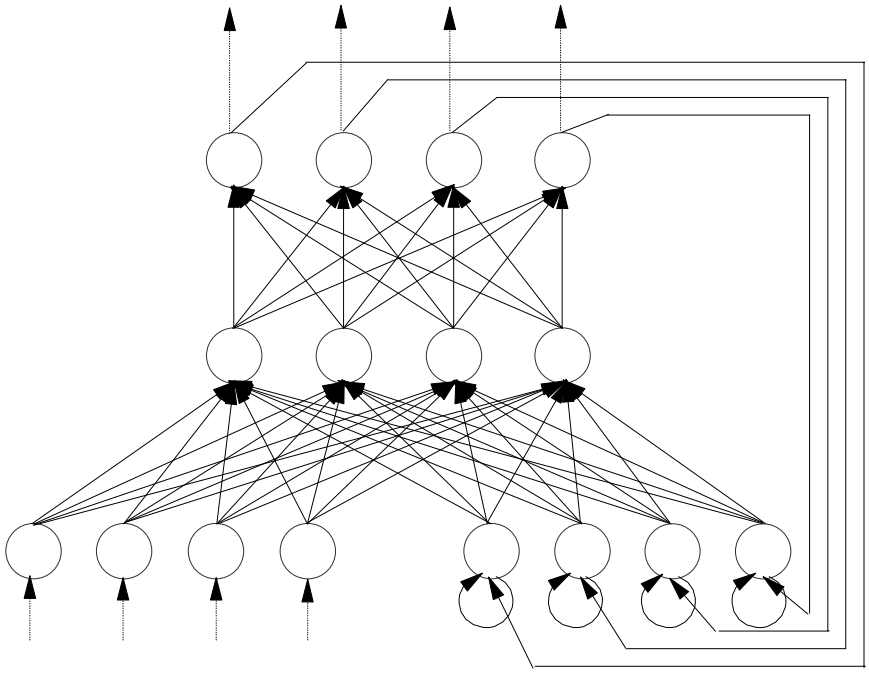


Figure 18 The sequence generator network.

The unmodified 2DELTA-GANN and the re-evaluation and no rules variants of 2DELTA-GANN as described in table 1 are tested for fifty independent runs. Results are presented in table 6 for:

- the number of successful runs.
- the lowest error $(2 \times \boldsymbol{E})$ of any run.
- the average error $(2 \times \boldsymbol{E})$ over the fifty runs.
- the fewest generations for any run.
- the average number of generations for all runs.
- the fewest evaluations for any run.
- the average evaluations for all runs.

A good result for these tests would show the number of successful runs at a maximum, and all other measures at a minimum.

Sequence Generator Network							
	Successful	$2 \times \boldsymbol{E}$		Generations		Evaluations	
Method	Runs	Lowest	Average	Fewest	Average	Fewest	Average
Re-run	9	0.2986	1.5888	39	178.8	19500	89400
Re-evaluation	33	0.1156	0.6554	32	120.2	31118	119834
No Rules	37	0.0475	0.6442	35	115.9	34166	115526
Back-Propagation	40	0.2156	0.7571	n/a	n/a	364	40962

Population = 500, Probability of Crossover = 0.2, Probability of Mutation = 0.005

Table 6 Results for the sequence generator network over 50 runs.

These results show quite clearly that the modifications made to the 2DELTA-GANN algorithm have improved its performance significantly with this type of recurrent network.

The improvement in success rate of the algorithm from *18%* to *74%* shows clearly that the method is very capable of successfully and consistently training this type of recurrent neural network.

The results show again that the removal of the application of the rules to update the delta values has improved the performance of the algorithm, although the large reduction in the lowest training error may indicate over-training.

These results show that while back-propagation in this case out-performed 2DELTA-GANN with regard to the training time, 2DELTA-GANN produced better results with regard to network accuracy. The much lower error value obtained by 2DELTA-GANN could be an indication that perhaps the network was over-trained by 2DELTA-GANN.

4.5.2 The Sine Network

The sine network shown in figure 19 is an example of a simple recurrent network. The task of the network is to predict the next number in a sequence of numbers, and in this test the sequence of numbers presented to the network are from the sine function. Numbers from the sequence are presented to the network one number at a time. This network uses the sigmoidal activation for the hidden layer, and the linear activation function for the output layer. Input to and output from the network are continuous. Input to the network was not normalised before processing. Training of the network is stopped when, for all input patterns, the output from the output unit is within *10%* of the expected output (the selection of *10%* was arbitrary); or after *7500* trials if this condition is not reached before then.

Recurrent back-propagation is also used to train this network, and the results are compared with those achieved by 2DELTA-GANN. The comparative results are presented in table 9.

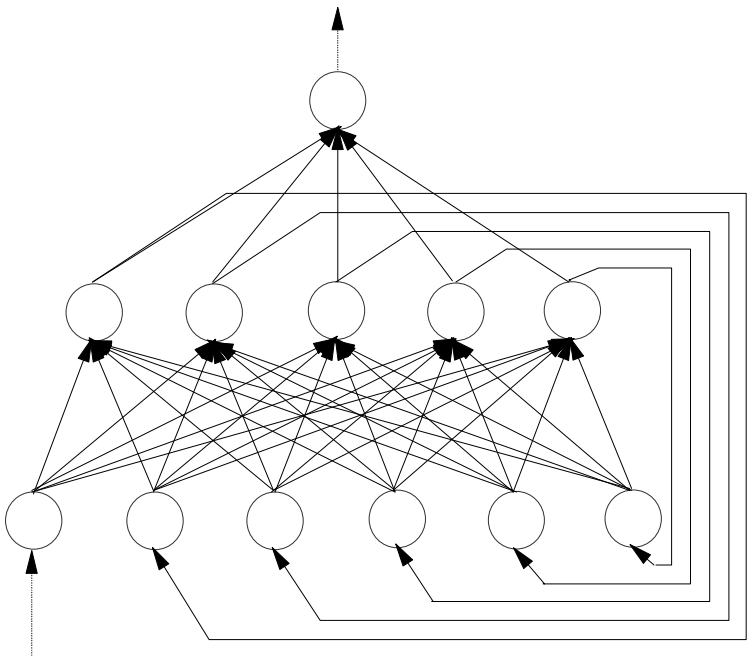


Figure 19 The sine network.

The recurrent network architecture and the recurrent back-propagation test program for the sine network are from [Tveter, 1997]. The data used was generated specifically for this test, and divided into a training set, a validation set and a test set. The validation set is used in the early stopping method when training the network using the recurrent back-propagation technique. The test set is a measure of the network's ability to generalise for unseen data. Performance of the networks trained using the 2DELTA-GANN method is also measured for the validation and test sets for comparison purposes. The validation data set is not used by either of the 2DELTA-GANN methods to stop training.

In order to determine if the uniform crossover method implemented by 2DELTA-GANN is a disruptive influence as a result of the crossover probability used, each of the methods is tested for a number of crossover probabilities.

The re-evaluation and no rules variants of the 2DELTA-GANN method as described in table 1 are tested for ten independent runs for several different crossover probabilities. For each crossover probability, tables 7 and 8 show results for:

- the lowest normalised error for the training data set achieved over the ten runs.
- the average normalised error for the training data set over the ten runs.
- the normalised error for the validation and test data sets *(for the networks which achieved the lowest normalised training error).*
- the number of evaluations for the training process.

A good result for these tests would show the values for each of the measures at a minimum.

Re-evaluation Method					
Crossover Probability	Training Set		Validation Set	Test Set	Evaluations
	Lowest	Average			
0.15	0.00334	0.00760	0.00321	0.00326	15000
0.25	0.00300	0.00477	0.00300	0.00300	15000
0.45	0.00171	0.00354	0.00172	0.00171	15000
0.65	0.00091	0.00278	0.00099	0.00095	15000
0.85	0.00185	0.00479	0.00187	0.00186	15000

Population = 100, Probability of Mutation = 0.005

Table 7 Results for the sine network for the re-evaluation method over 10 runs.

No Rules Method					
Crossover Probability	Training Set		Validation Set	Test Set	Evaluations
	Lowest	Average			
0.15	0.00160	0.00330	0.00155	0.00157	15000
0.25	0.00045	0.00367	0.00049	0.00046	15000
0.45	0.00085	0.00225	0.00076	0.00079	15000
0.65	0.00098	0.00182	0.00091	0.00094	15000
0.85	0.00045	0.00270	0.00058	0.00049	15000

Population = 100, Probability of Mutation = 0.005

Table 8 Results for the sine network for the no rules method over 10 runs.

These results indicate that for the sine network, the random application of the delta values has increased performance over the standard 2DELTA-GANN algorithm slightly.

The results also indicate that the probability of crossover does affect the performance of both variants of the 2DELTA-GANN algorithm, although not in any uniform or predictable way. The fact that the probability of crossover actually affects the no rules method indicates that the crossover of the delta values is affecting the outcome. Whether this just a macro-mutation effect (crossover so disruptive so as to cause a mutation-like effect) is not yet evident.

It is noted that in each case the training error achieved is very similar to both the validation and test errors. This would tend to indicate that the network is not overtrained and that a good stopping point was chosen for training.

The best runs of both the re-evaluation and the no rules method are compared with the results from ten runs of the recurrent back-propagation method from [Tveter, 1997]. The recurrent back-propagation runs were stopped after *30000* epochs if no result had been achieved before then. These results, shown in table 9, indicate that the 2DELTA-GANN method out-performs recurrent back-propagation for the sine network in terms of both training time and prediction accuracy.

Comparison of 2DELTA-GANN with Recurrent Back-Propagation						
	Training Set		Validation	Test	Evaluations	
Method	Lowest	Average	Set	Set	Fewest	Average
Re-evaluation	0.00091	0.00278	0.00099	0.00095	15000	15000
No Rules	0.00045	0.00270	0.00049	0.00046	15000	15000
Recurrent Back-Propagation	0.00096	0.04178	0.00089	0.00091	15006	31630

Table 9 Comparison of 2DELTA-GANN with Recurrent Back-Propagation for the sine network.

4.5.3 The Sunspot Networks

4.5.3.1 Simple Recurrent Network

The simple recurrent network shown in figure 20 is trained to predict the number of sunspots expected to be seen in any year. The sunspot data is well known, having been recorded since at least *1700* . A graph of sunspot activity since the year *1700* shows that regular fluctuations seem to take place over a *10* to *12* year period, so *12* input units were used. Data is presented to these inputs by the use of a moving window across the sunspot data. The task of the network is to predict the next number in a sequence of numbers.

This network uses the sigmoidal activation for both the hidden and output layers. Input to and output from the network are continuous. Input to the network was normalised before processing. Training of the network is stopped when, for all input patterns, the output from

the output unit is within *10%* of the expected output (the selection of *10%* was arbitrary); or after *50000* trials if this condition is not reached before then.

The 2DELTA-GANN results are compared with published results for the recurrent back-propagation method from [McCluskey, 1993]. McCluskey trained a simple recurrent network of the same form to predict the sunspot data. The only difference between the network trained by 2DELTA-GANN and that used by McCluskey is that McCluskey's network has a bias input unit with a fixed weight of *0.5*. The network trained by 2DELTA-GANN has no such bias input unit with a fixed weight, but allows the bias values of the hidden and output units to be trained along with the connection weights.

The sunspot data used is from [Tong, 1991]. For consistency with McCluskey, the sunspot data from *1700* to *1920* is used as the training set, and the data from *1921* to *1979* as the test set. The test set is further divided into an early test set and a late test set for consistency with McCluskey. The early test set includes the years *1921* to *1955*, and the late test set the years *1956* to *1979*. The weights from the best result achieved after training are used to determine the performance on the test set. The comparison with McCluskey's results is shown in table 12.

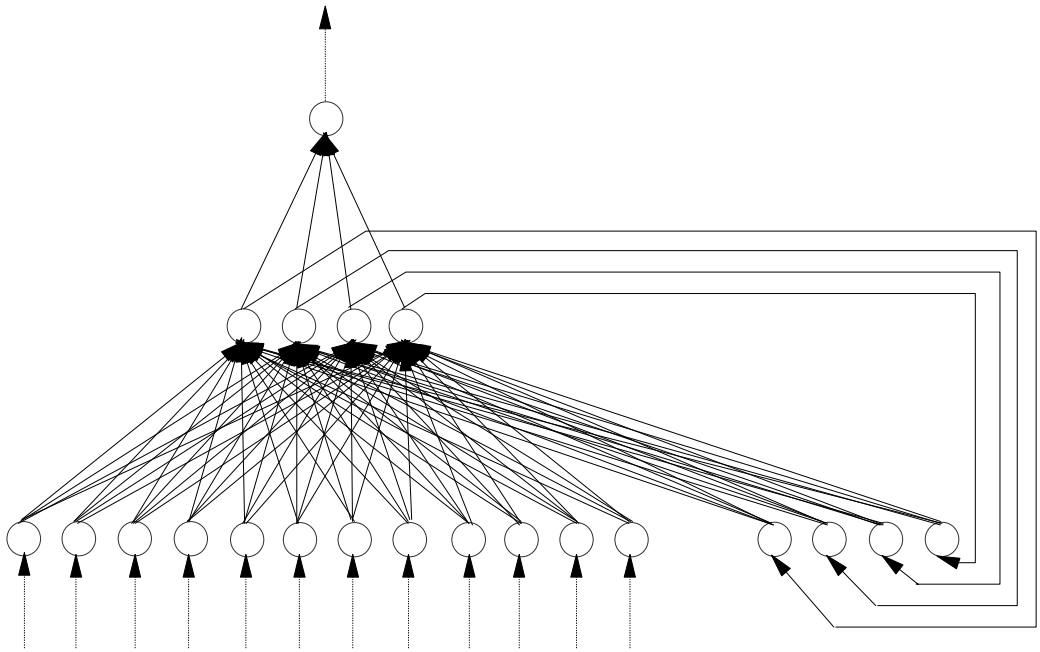


Figure 20 The sunspot simple recurrent network.

Both the re-evaluation method and the no rules method as described in table 1 are used to train this network. Each method is tested for five independent runs (for consistency with McCluskey). Results are presented for:

- the lowest normalised error achieved over the five runs for the training data set.
- the average normalised error over the five runs for the training data set.
- the normalised error for the test data sets *(for the networks which achieved the lowest normalised training error).*
- the number of evaluations for the training process.

A good result for these tests would show the values for each of the measures at a minimum.

As for the sine network, each of the methods is tested for a number of crossover probabilities. The results for the two methods are presented in tables 10 and 11.

Re-evaluation Method						
Crossover Probability	Training Set		Test Set			Evaluations
	Lowest	Average	Full	Early	Late	
0.15	0.1380	0.1544	0.1436	0.1172	0.2670	100000
0.25	0.1347	0.1496	0.1343	0.1180	0.1937	100000
0.45	0.1233	0.1343	0.1263	0.1135	0.5210	100000
0.65	0.1267	0.1359	0.1359	0.1019	0.1937	100000
0.85	0.1426	0.1498	0.1830	0.1032	0.4721	100000

Population = 1000, Probability of Mutation = 0.005

Table 10 Results for sunspot SRN for the re-evaluation method over 5 runs.

No Rules Method						
Crossover Probability	Training Set		Test Set			Evaluations
	Lowest	Average	Full	Early	Late	
0.15	0.1301	0.1454	0.1594	0.1358	0.2394	100000
0.25	0.1242	0.1347	0.1445	0.1330	0.7080	100000
0.45	0.1274	0.1368	0.1551	0.1416	0.2811	100000
0.65	0.1294	0.1373	0.1565	0.1387	0.2704	100000
0.85	0.1190	0.1394	0.1137	0.0996	0.2347	100000

Population = 1000, Probability of Mutation = 0.005

Table 11 Results for sunspot SRN for the no rules method over 5 runs.

These results indicate that the removal of the rules governing the application of the delta values has had little effect on the performance of 2DELTA-GANN. The re-evaluation technique has performed marginally better when comparing the training errors, but it can be seen that the no rules method has produced networks which generalise better. As with the results presented for the sine network, although the effect of varying the crossover probability can be seen, this effect is not uniform or predictable. The effect of varying the crossover probability is not as pronounced as with the sine network.

The best runs of both the re-evaluation and the no rules method are compared with published results for the recurrent back-propagation method from [McCluskey, 1993]. These results, shown in table 12, indicate that although the 2DELTA-GANN method compares favourably with recurrent back-propagation with regard to the prediction accuracy of the resultant network, it is not as efficient when training time is compared.

Comparison of 2DELTA-GANN with Recurrent Back-Propagation					
		Test Set			
Method	Training Set	Full	Early	Late	Evaluations
Re-evaluation	0.123	0.126	0.114	0.521	100000
No Rules	0.119	0.114	0.100	0.235	100000
Recurrent Back-Propagation	0.102	0.152	0.091	0.190	10000

Table 12 Comparison of 2DELTA-GANN with Recurrent Back-Propagation for the sunspot SRN.

It should be noted that the run considered the best run for these purposes is that which has produced the lowest error on the training set. It can be seen that this run has not necessarily produced the network which generalises the best, indicating that the network may have been over-trained by 2DELTA-GANN.

4.5.3.2 Real Time Recurrent Network

The real-time recurrent network shown in figure 21 is also trained to predict sunspot activity. This network uses the sigmoidal activation for the processing layer (combined hidden and output layers). Input to and output from the network are continuous. Input to the network was normalised before processing. As for the simple recurrent network, training of the network is stopped when, for all input patterns, the output from the output unit is within *10%* of the expected output (the selection of *10%* was arbitrary); or after *50000* trials if this condition is not reached before then.

The 2DELTA-GANN results are compared with published results for the real time recurrent learning method from [McCluskey, 1993]. As with the simple recurrent network, the difference between the network trained by 2DELTA-GANN and that used by McCluskey is that McCluskey's network has a bias input unit with a fixed weight of *0.5*, whereas 2DELTA-GANN allows the bias values of the hidden and output units to be trained along with the connection weights. The comparison is shown in table 15.

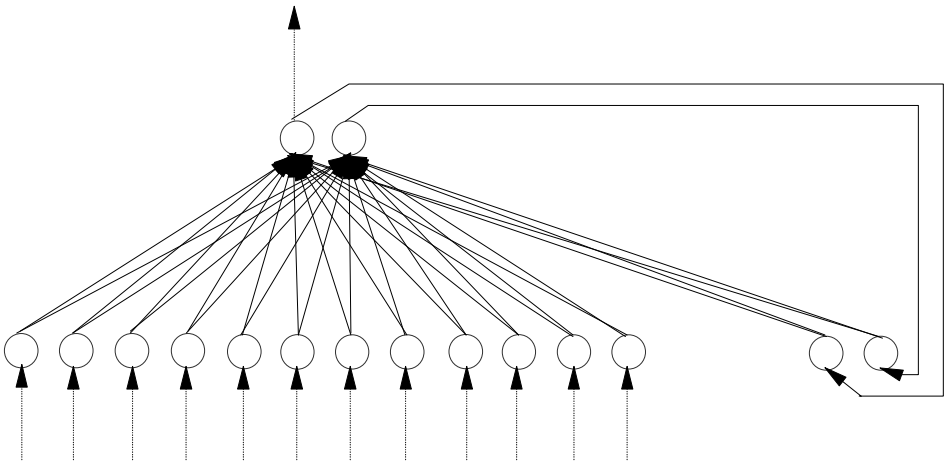


Figure 21 The sunspot real time recurrent network.

Both the re-evaluation method and the no rules method as described in table 1 are used to train this network. Each method was tested for five independent runs (for consistency with McCluskey). Results are presented for:

- the lowest normalised error achieved over the five runs for the training data set.
- the average normalised error over the five runs for the training data set.
- the normalised error for the test data sets *(for the networks which achieved the lowest normalised training error).*
- the number of evaluations for the training process.

A good result for these tests would show the values for each of the measures at a minimum.

Again, each of the methods is tested for a number of crossover probabilities. The results for each method are presented in tables 13 and 14.

Re-evaluation Method						
Crossover Probability	Training Set		Test Set			Evaluations
	Lowest	Average	Full	Early	Late	
0.15	0.1592	0.1642	0.1479	0.1140	0.2747	100000
0.25	0.1522	0.1601	0.1844	0.1358	0.3684	100000
0.45	0.1604	0.1650	0.1546	0.1103	0.4298	100000
0.65	0.1569	0.1633	0.1641	0.1397	0.2891	100000
0.85	0.1631	0.1656	0.1740	0.1424	0.2832	100000

Population = 1000, Probability of Mutation = 0.005

Table 13 Results for sunspot RTRN for the re-evaluation method over 5 runs.

No Rules Method						
Crossover Probability	Training Set		Test Set			Evaluations
	Lowest	Average	Full	Early	Late	
0.15	0.1570	0.1617	0.1549	0.1208	0.2343	100000
0.25	0.1537	0.1609	0.1863	0.1694	0.3192	100000
0.45	0.1555	0.1586	0.1954	0.1616	0.3211	100000
0.65	0.1551	0.1588	0.1558	0.1353	0.4302	100000
0.85	0.1578	0.1633	0.1507	0.1263	0.2952	100000

Population = 1000, Probability of Mutation = 0.005

Table 14 Results for sunspot RTRN for the no rules method over 5 runs.

These results show again that the removal of the rules governing the application of the delta values has had little effect on the performance of 2DELTA-GANN. The results also indicate that varying the probability of crossover has not affected the results significantly.

The performance of 2DELTA-GANN is compared with published results for the real time recurrent learning method from [McCluskey, 1993]. These results, shown in table 15, indicate that as for the simple recurrent network, although the 2DELTA-GANN method compares reasonably well with real time recurrent learning with regard to the prediction accuracy of the resultant network, it is not as efficient when training time is compared.

Comparison of 2DELTA-GANN with Real Time Recurrent Learning					
Method	Training Set	Test Set			Evaluations
		Full	Early	Late	
Re-evaluation	0.152	0.184	0.136	0.368	100000
No Rules	0.153	0.186	0.169	0.319	100000
Real Time Recurrent Learning	0.183	0.133	0.126	0.138	10000

Table 15 Comparison of 2DELTA-GANN with Real Time Recurrent Learning for the sunspot RTRN.

4.5.4 Stopping Training

In order to prevent over-fitting with the 2DELTA-GANN technique, a method of early stopping is tested. With this method, the training data is divided into a training set and a validation set, and the best weights from each generation are used to calculate the network error for the validation set. Training is stopped if the error for the validation set begins to rise. This method is described in [Sarle, 1995]. The error for the validation data set is an estimate of the quality of the network and its ability to generalise for unseen data. The goal of the early stopping method is to maximise the quality of the network and its ability to generalise.

Tests are run for the sunspot simple recurrent network using the no rules method as described in table 1. Ten independent runs are conducted, each using a crossover probability of *0.45*. For these tests training is stopped if the error for the validation set at any generation exceeded the best error for the validation set by *15%* or more (the selection of *15%* was arbitrary). Results are presented in table 16.

No Rules Method with Early Stopping			
Run	**Training Set**	**Validation Set**	**Evaluations**
1	0.1436	0.1158	74000
2	0.1274	0.1306	94000
3	0.1547	0.1699	64000
4	0.1477	0.1340	58000
5	0.1523	0.1343	52000
6	0.1397	0.1256	80000
7	0.1314	0.1162	74000
8	0.1588	0.1524	70000
9	0.1319	0.1321	92000
10	0.1650	0.1536	50000

Population = 1000, Probability of Crossover = 0.45, Probability of Mutation = 0.005

Table 16 Early Stopping results for the sunspot SRN.

It can be seen from these results that early stopping can help to reduce the training time of the 2DELTA-GANN method, though perhaps not as significantly or consistently as desired. Although these results are encouraging, and this method of early stopping is possibly useful, in its current form it is considered to be more appropriate for gradient descent techniques which move in a more incremental fashion towards some minimum. The genetic search implemented by the 2DELTA-GANN method is less likely to move towards a minimum in such an incremental fashion, so further investigation of and modifications to this method of early stopping are necessary.

The comparison of the early stopping results with those obtained by the no rules method without early stopping and back-propagation is presented in table 17. For this comparison the validation set used by the early stopping method is the full test set used by the other methods. These results show that the implementation of early stopping has enabled 2DELTA-GANN to produce a network which generalises better than either of the other techniques. While the training time is still not competitive with back-propagation, these results are encouraging.

Comparison of Early Stopping Results			
Method	**Training Set**	**Test Set**	**Evaluations**
No Rules	0.127	0.155	100000
No Rules with Early Stopping	0.127	0.131	94000
Recurrent Back-Propagation	0.102	0.152	10000

Population = 1000, Probability of Crossover = 0.45, Probability of Mutation = 0.005

Table 17 Comparison of early stopping results for sunspot SRN.

Chapter 5
Conclusions and Further Work

5.1 Conclusions

Specific goals of this research were:

- To test, verify, and improve the original implementation of 2DELTA-GANN.

  The 2DELTA-GANN method as implemented by Krishnan was analysed and found to have some minor defects in its implementation. Those defects were repaired and the method re-tested on a selection of the feed-forward networks tested in the original work by Krishnan. The results achieved by the repaired code were shown to be as good or better than those achieved by the original 2DELTA-GANN.

- To test the hypothesis that the evolution of the rules provides little or no benefit to the overall 2DELTA-GANN algorithm, and to show that the worth of the technique is in the usefulness of genetic recombination as it is applied to the delta values themselves.

  This work shows that the concept of using delta values to move the network weights incrementally towards a solution has merit, but the attempt to evolve heuristic rules to govern the application of those delta values does not. An important outcome of this work is that it shows an evolutionary approach to training neural networks, both feed-forward and recurrent, is viable, and in some circumstances preferable to gradient descent techniques. The 2DELTA-GANN technique would be most useful for training feed-forward and recurrent neural networks that are difficult to train by existing gradient descent techniques.

  The results of this work suggest that the usefulness of the technique is actually in the search being conducted by the selection and recombination of the real-valued delta values rather than the bit string encoding of the rules used to apply those delta values.

- To investigate the possibility of extending the technique to include recurrent neural networks, and attempt to develop the method to the point that it compares favourably with the more commonly used gradient descent techniques.

  The neural network evaluation code which comprised the fitness function of 2DELTA-GANN was redesigned and re-coded to allow training of several different types of recurrent neural network. A range of problems for several different network architectures were then tested with the new 2DELTA-GANN method.

  The results from these tests are encouraging in that they indicate that the evolutionary method implemented by 2DELTA-GANN is capable of training different classes of feed-forward and recurrent neural networks with some consistency.

The new 2DELTA-GANN was tested on a simple recurrent network to predict a series of values for the sine function, and its performance compared against recurrent back-propagation for the same network. 2DELTA-GANN was shown to out-perform recurrent back-propagation for this test.

2DELTA-GANN was also tested on both a simple recurrent network and a real time recurrent network to predict the average annual number of sunspots, and its performance compared against results published for the same tests on similar network architectures. 2DELTA-GANN was shown to produce networks comparable to recurrent back-propagation and real time recurrent learning with regard to network performance and accuracy. Training time for 2DELTA-GANN was shown to be somewhat greater than the gradient descent techniques.

- To investigate and evaluate a method of stopping training to prevent over-fitting by the 2DELTA-GANN technique.

  A method of early stopping by the use of a validation data set was investigated. The method investigated uses a validation data set, different from the training data set, to monitor the performance of the network being evolved. After each generation the best performing individual is evaluated using the validation data set, and if the network error for the validation data set increases from the minimum by some arbitrary amount, training is stopped.

  Results suggest that the technique has the potential to help to prevent the genetic algorithm over-training the neural network. The tests conducted show that by monitoring the network error for the validation data set, training time can be reduced and a network capable of better generalisation produced.

The overall goal of this research was to investigate the viability of utilising genetic algorithms to determine the network weights and biases for fixed architecture feed-forward and recurrent neural networks, and to compare this method with existing, more common methods for training artificial neural networks.

The 2DELTA-GANN method of training neural networks by the use of genetic algorithms was tested and extended to include recurrent networks. The results of this work indicate that the method is very promising, especially for hard problems which are not easily solved by current back-propagation methods.

5.2 Further Work

Some obvious further work is in the gene structure and role of the rule bits for this method. Since the results presented in this work suggest that the rules and their evolution have little if any real impact on the performance of the search, some investigation into the jettisoning, or radical modification of those rules should be conducted, and perhaps a new approach taken. Parts of this method have merit, as evidenced by the results achieved, and those parts need to be retained and enhanced.

Krishnan suggested in his thesis that the dominant genetic operator at work in this method is actually mutation. While there was a coding defect in the original 2DELTA-GANN code which prevented crossover of the rule bits, the results which have been presented in this work show that this probably would not have led Krishnan to a different conclusion. As suggested earlier, crossover of the real-valued delta values seems to be beneficial to the outcome of the genetic search, but further investigation of the disruptive effects of uniform crossover and whether it is in fact just causing a macro-mutation effect would be useful. Some investigation into the implementation of other crossover techniques could also be useful.

Investigation into population based incremental learning (PBIL) [Baluja, 1995] could be instructive. In [Baluja, 1995], Baluja and Caruana describe a variation of the genetic algorithm in which the crossover operator is abstracted away. An investigation of how this could be related to the results being produced by 2DELTA-GANN may be useful.

As part of any further investigation into the disruptive effects of uniform crossover, further investigation into the application of Goldberg's work with virtual alphabets in [Goldberg, 1991] could also be instructive. In that work Goldberg postulates that for a real-coded genetic algorithm, *"selection dominates early GA performance and restricts subsequent search to intervals with above-average function value"* [Goldberg, 1991, p1]. Goldberg further suggests that those intervals *"may be further subdivided on the basis of their attraction under genetic hillclimbing"* [Goldberg, 1991, p1]. Goldberg refers to the subintervals as virtual characters, and the collection of characters along a given dimension he calls a virtual alphabet. According to Goldberg, it is the virtual alphabet that is searched during the recombination, and this is sufficient in many problems to find good solutions.

The tests for both the feed-forward and recurrent networks highlight again that this method is sensitive to the selection of the initial bounds of the delta values. This method has also shown to be sensitive to several other parameters of the genetic algorithm, such as population size, probability of mutation and crossover, and initial values of the population to name just a few. This suggests some useful further work to investigate the feasibility of implementing a meta-level genetic algorithm to optimise the parameters for 2DELTA-GANN.

Some further work with both feed-forward and recurrent neural networks which existing gradient descent techniques find difficult to train would be instructive as to the usefulness of the 2DELTA-GANN technique.

Further investigation into a reliable method of early stopping would be useful. This would produce benefits both by helping to develop networks more capable of generalising, and by reducing the training time of the 2DELTA-GANN technique.

Chapter 6
Bibliography

Amaral, 1995 Amaral, J. N., Turner, K., and Ghosh, J. 1995.
Designing Genetic Algorithms for the State Assignment Problem.
In *IEEE Transactions on Systems, Man and Cybernetics, April 1995.*

Angeline, 1994 Angeline, P. J., Saunders, G. M., and Pollack, J. B. 1994.
An Evolutionary Algorithm that Constructs Recurrent Neural Networks.
In *IEEE Transactions on Neural Networks, Vol. 5, No. 1, 1994.*

Balakrishnan, 1995 Balakrishnan, K., and Honavar, V. 1995.
Evolutionary Design of Neural Architectures - A Preliminary Taxonomy and Guide to Literature.
Technical Report CS TR #95-01, Artificial Intelligence Group, Iowa State University, Iowa, USA.

Baluja, 1995 Baluja, S. and Caruana, R. 1995.
Removing the Genetics from the Standard Genetic Algorithm.
Technical Report CMU-CS-95-141, Department of Computer Science, Carnegie Mellon University, Pennsylvania, USA.

Belew, 1990 Belew, R. K., McInerney, J., and Schraudolph, N. N. 1990.
Evolving Networks: using the Genetic Algorithm with Connectionist Learning.
CSE Technical Report CS90-174, Computer Science and Engineering Department, University of California at San Diego, California, USA.

Caruana, 1989 Caruana, R. A,. Schaffer, J. D., and Eshelman, L. J. 1989.
An Experimental Comparison of Various Crossover Operators.
In *Proceedings of the Third International Conference on Genetic Algorithms (ICGA-89).*

Elman, 1991 Elman, J. L. 1991.
Distributed Representations, Simple Recurrent Networks, and Grammatical Structure.
In *Machine Learning 7, 1991*

Esparcia-Alcázar, 1997 Esparcia-Alcázar, A., and Sharman, K. 1997.
Evolving Recurrent Neural Network Architectures by Genetic Programming.
In *Genetic Programming 1997: Proceedings of the Second Annual Conference.*

Fahlman, 1988
Fahlman, S. E. 1988.
An Empirical Study of Learning Speed in Backpropagation Networks.
Technical Report CMU-CS-88-162, Department of Computer Science, Carnegie Mellon University, Pennsylvania, USA.

Fahlman, 1990
Fahlman, S. E., and Lebiere, C. 1990.
The Cascade-Correlation Learning Architecture.
In *Advances in Neural Information Processing Systems 2, 1990.*

Fogel, 1966
Fogel, L. J., Owens, A. J., and Walsh, M. J. 1966.
Artificial Intelligence Through Simulated Evolution.
John Wiley and Sons, NY.

Fullmer, 1991
Fullmer, B., and Miikkulainen, R. 1991.
Using Marker-based Genetic Encoding of Neural Networks to Evolve Finite-State Behaviour.
In *Proceedings of the First European Conference on Artificial Life (ECAL-91).*

Geman, 1992
Geman, S., Bienenstock, E., and Doursat, R. 1992.
Neural Networks and the Bias/Variance Dilemma.
In *Neural Computation 4, 1992.*

Goldberg, 1989
Goldberg, D. E. 1989.
Genetic Algorithms in Search, Optimization & Machine Learning.
Addison-Wesley, Reading, MA.

Goldberg, 1991
Goldberg, D. E. 1991.
Real-coded Genetic Algorithms, Virtual Alphabets, and Blocking.
In *Complex Systems 5, 1991.*

Grefenstette, 1984
Grefenstette, J. J. 1984.
A User's Guide to GENESIS.
Technical Report CS-83-11, Computer Science Department, Vanderbilt University, Tennessee, USA.

Holland, 1975
Holland, J. H. 1975.
Adaptation in Natural and Artificial Systems.
Ann Arbor: The University of Michigan Press.

Jordan, 1986
Jordan, M. I. 1986.
Attractor Dynamics and Parallelism in a Connectionist Sequential Machine.
In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society. Lawrence Erlbaum, 1986.*

Korning, 1995
Korning, P. G. 1995.
Training Neural Networks by means of Genetic Algorithms Working on Very Long Chromosomes.
In *The International Journal of Neural Systems, Vol. 6 No. 3 (September 1995).*

Koza, 1992
Koza, J. R. 1992.
Genetic Programming: On the Programming of Computers by Means of Natural Selection.
MIT Press, Cambridge, MA.

Krishnan, 1994a
Krishnan, R. 1994.
2DELTA-GANN: A New Method of Training Neural Networks Using Genetic Algorithms.
Master of Applied Science thesis, Department of Computer Science, Royal Melbourne Institute of Technology, Victoria, Australia.

Krishnan, 1994b
Krishnan, R., and Ciesielski, V. B. 1994.
2DELTA-GANN: A New Approach To Training Neural Networks Using Genetic Algorithms.
In *Proceedings of the Fifth Australian Conference on Neural Networks, 1994.*

McClelland, 1988
McClelland, J. L., and Rumelhart, D. E. 1988.
Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises.
MIT Press, Cambridge, MA.

McCluskey, 1993
McCluskey, P. C. 1993.
Feedforward and Recurrent Neural Networks and Genetic Programs for Stock Market and Time Series Forecasting.
Master of Applied Science thesis, Department of Computer Science, Brown University, Rhode Island, USA.

McCulloch, 1943
McCulloch, W., and Pitts, W. 1943.
A Logical Calculus of The Ideas Immanent in Nervous Activity.
Bulletin of Mathematical Biophysics 1943.

Miller, 1989
Miller, G. F., Todd, P. M., and Hegde, S. U. 1989.
Designing Neural Networks using Genetic Algorithms.
In *Proceedings of the Third International Conference on Genetic Algorithms (ICGA-89).*

Mitchell, 1996
Mitchell, M. 1996
An Introduction to Genetic Algorithms.
MIT Press, Cambridge, MA.

Montana, 1989
Montana, D. J., and Davis, L. D. 1989.
Training Feedforward Networks using Genetic Algorithms.
In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-89)*.

Moriarty, 1993
Moriarty, D., and Miikkulainen, R. 1993.
Evolving Complex Othello Strategies using Marker-based Genetic Encoding of Neural Networks.
Technical Report AI93-206, Department of Computer Sciences, The University of Texas at Austin, Texas, USA.

Moriarty, 1996a
Moriarty, D., and Miikkulainen, R. 1996.
Efficient Reinforcement Learning through Symbiotic Evolution.
In *Machine Learning 22, 1996.*

Moriarty, 1996b
Moriarty, D., and Miikkulainen, R. 1996.
Hierarchical Evolution of Neural Networks.
Technical Report AI96-242, Department of Computer Sciences, The University of Texas at Austin, Texas, USA.

Moriarty, 1997
Moriarty, D. E. 1997.
Symbiotic Evolution of Neural Networks in Sequential Decision Tasks.
PhD thesis, The University of Texas at Austin, Texas, USA.

Pineda, 1988
Pineda, F. J. 1988.
Dynamics and Architecture for Neural Computation.
In *Journal of Complexity 4, 1988.*

Potter, 1992
Potter, M. A. 1992.
A Genetic Cascade-Correlation Learning Algorithm.
In *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92).*

Radcliffe, 1990
Radcliffe, N. J. 1990.
Genetic Neural Networks on MIMD Computers.
PhD. thesis, Physics Department, University of Edinburgh, Scotland.

Radcliffe, 1991
Radcliffe, N. J. 1991.
Genetic Set Recombination and its Application to Neural Network Topology Optimisation.
Technical Report EPCC-TR-91-21, University of Edinburgh, Scotland.

Rosenblatt, 1962
Rosenblatt, F. 1962.
Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.
Spartan Books, NY.

Rumelhart, 1986
Rumelhart, D. E., and McClelland, J. L. eds. 1986.
Parallel Distributed Processing: Explorations in The Microstructure of Cognition.
MIT Press, Cambridge, MA.

Sarle, 1995
Sarle, W. S. 1995.
Stopped Training and Other Remedies for Overfitting.
In *Proceedings of the 27th Symposium on the Interface, 1995.*

Schraudolph, 1990
Schraudolph, N. N., and Belew, R. K. 1990.
Dynamic Parameter Encoding for Genetic Algorithms.
Technical Report LAUR 90-2795, Los Alamos National Laboratory Center for Nonlinear Studies, Los Alamos, NM.

Schraudolph, 1992
Schraudolph, N. N., and Grefenstette, J. J. 1992.
A User's Guide to GAUCSD 1.4, July 1992.
Technical Report CS92-249, Computer Science and Engineering Department, University of California, San Diego, California, USA.

Spears, 1991
Spears, W. M., and DeJong, K. A. 1991.
On the Virtues of Parameterized Uniform Crossover.
In *Proceedings of the Fourth Annual Conference on Genetic Algorithms (ICGA-91).*

Syed, 1995
Syed, O. 1995.
Applying Genetic Algorithms to Recurrent Neural Networks for Learning Network Parameters and Architecture.
Master of Science thesis, Department of Electrical Engineering, Case Western Reserve University, Ohio, USA.

Tong, 1991
Tong, H. 1991.
Non-Linear Time Series: A Dynamical Systems Approach.
Oxford University Press.

Tveter, 1997
Tveter, D. R. 1997.
The Pattern Recognition Basis of Artificial Intelligence.
IEEE Computer Society Press, Available December 1997
(Example code and networks retrieved from http://www.mcs.com/~drt/svbp.html)

Werbos, 1974
Werbos, P. J. 1974.
Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.
PhD thesis, Harvard University, Cambridge, MA, USA. Reprinted in P. Werbos, The Roots of Backpropagation: From Ordered Derivatives to Neural Networks to Political Forecasting. Wiley, 1993.

Whitley, 1989 Whitley, D., and Hanson, T. 1989.
Optimizing Neural Networks Using Faster More Accurate Genetic Search.
In *Proceedings of the Third International Conference on Genetic Algorithms (ICGA-89).*

Whitley, 1991 Whitley, D., Dominic, S. and Das, R. 1991.
Genetic Reinforcement Learning with Multilayered Neural Networks.
In *Proceedings of the Fourth International Conference on Genetic Algorithms (ICGA-91).*

Williams, 1989 Williams, R. J., and Zipser, D. 1989.
A Learning Algorithm for Continually Running Fully Recurrent Neural Networks.
In *Neural Computation 1, 1989.*

Yao, 1997 Yao, X., and Liu, Y. 1997
A New Evolutionary System for Evolving Artificial Neural Networks.
In *IEEE Transaction on Neural Networks, Vol 8, No. 3, 1997.*

Appendix A
Genetic Algorithm Parameter Details

2DELTA-GANN was developed using the GAUCSD genetic algorithm research package [Schraudolph, 1992]. GAUCSD was based on the GENESIS genetic algorithm research package, developed by John Grefenstette [Grefenstette, 1984].

GAUCSD is implemented as a generational replacement genetic algorithm. With this style of algorithm two populations are maintained: the population which describes the current generation, and the population which replaces the current population and which describes the next generation. The genetic operators are applied to the individuals of the current population to generate the population describing the next generation.

2DELTA-GANN was implemented using a fitness scaling approach for the selection operator; a modified version of uniform crossover (catering for the modified gene structure); and a modified version of single bit mutation (again, catering for the modified gene structure).

The genetic algorithm is driven by parameters supplied to the package by way of an input file. To accommodate the changes made to the package for 2DELTA-GANN, the input file was modified and extended. A description of the parameters contained within the modified input file follows.

An explanation of the genetic algorithm parameters can be found in [Schraudolph, 1992]. The parameters which were added or modified for this thesis are:

Crossover Points this parameter was removed.

Init Weight Range this determines the bounds of the initial weights for the neural network Any weight w will be in the range

$$-\frac{InitWeightRange}{2} \le w \le +\frac{InitWeightRange}{2}$$

Max Weight Range this determines the bounds of the network weights and biases, as well as the bounds of both *delta1* and *delta2*. Any weight or delta value w will be bounded by

$$-MaxWeightRange \le w \le +MaxWeightRange$$

Delta1Range, Delta2Range these determine the bounds of the initial values of *delta1* and *delta2*. Any *delta1* value w will be bounded by

$$-\frac{Delta1Range}{2} \le w \le +\frac{Delta1Range}{2}$$

and *delta2* values are similarly bounded by Delta2Range.

Continuous Output set to *1* if the output of the network is continuous, *0* if it is binary.

Cont. Tolerance if negative, this number specifies the acceptable absolute difference between the expected output and the actual output of a unit, otherwise it specifies the acceptable difference as a fraction of the expected output. For example, a value of *–0.03* indicates that the actual output may differ from the expected output by up to and including *0.03* and still be considered correct, and a value of *0.1* indicates that the actual output may differ from the expected output by up to (and including) *10%* of the expected output and still be considered correct.

Loopback Weight The fixed value of a self-recurrent loopback weight on context units for recurrent networks (refer to the sequence generator network).

Normalise set to *1* if input data is to be normalised, *0* if it is not.

Output Layer set to *S* for sigmoidal activation, *L* for linear.

Early Stop set to *0.0* for no early stopping. If greater than *0.0*, this number specifies the percentage increase over the minimum validation error the validation error may attain before training is terminated.

Appendix B
Network Descriptions

Following is a description of each of the networks trained by 2DELTA-GANN for this thesis.

The description includes:

- a diagram of the network architecture
- a description of the training data and expected output
- the genetic algorithm input file for the network

B.1 The XOR Network

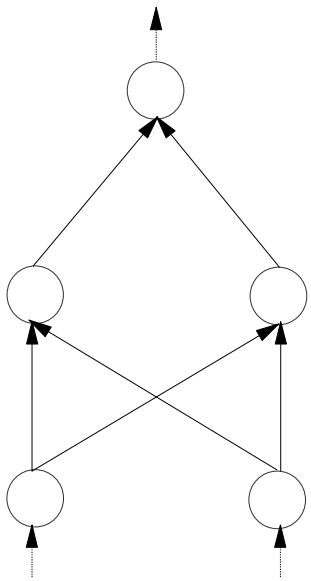


Figure 22 The XOR network.

The XOR network shown in figure 22 was trained with the training input vectors shown in table 18 to produce the expected output shown.

Input Vector	Expected Output
00	0
01	1
10	1
11	0

Table 18 XOR network expected operation.

The genetic algorithm input file for the XOR network:

Parameter	Value
Experiments	50
Total Trials	8000
Population Size	200
Structure Length	30
Crossover Rate	1.0
Mutation Rate	0.005
Generation Gap	0.9
Scaling Window	-1
Report Interval	1
Structures Saved	0
Max Gens w/o Eval	2
Dump Interval	0
Dumps Saved	0
Options	Acle
Random Seed	3237849206
Maximum Bias	0.99
Max Convergence	60
Conv Threshold	0.67
DPE Time Constant	0
Sigma Scaling	4.0
Init Weight Range	1.0
Max Weight Range	100.0
Crossover Prob	0.2
Delta1Range*	1.75
Delta2Range*	0.6
Continuous Output	0
Cont. Tolerance	0.0
Loopback Weight	0.0
Normalise	0
Output Layer	S
Early Stop	0.0

* (100.0, 100.0) best for re-evaluation (see table 2)

Table 19 XOR network input file.

B.2 The 4-2-4 Encoder Network

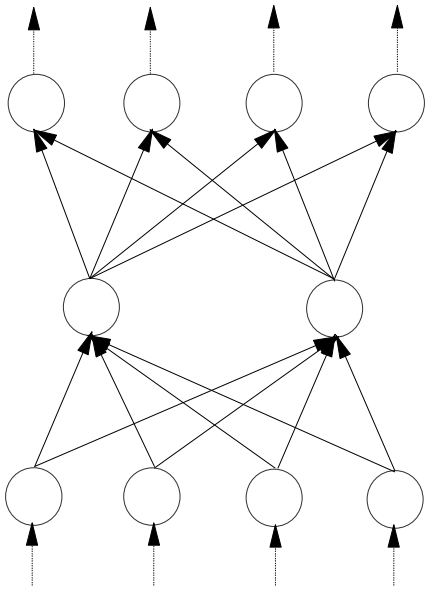


Figure 23 The 4-2-4 encoder network.

The 4-2-4 encoder network shown in figure 23 was trained with the training input vectors shown in table 20 to produce the expected output shown.

Input Vector	Expected Output
1000	1000
0100	0100
0010	0010
0001	0001

Table 20 4-2-4 encoder network expected operation.

The genetic algorithm input file for the 4-2-4 encoder network:

Parameter	Value
Experiments	50
Total Trials	10000
Population Size	200
Structure Length	30
Crossover Rate	1.0
Mutation Rate	0.005
Generation Gap	0.9
Scaling Window	-1
Report Interval	1
Structures Saved	0
Max Gens w/o Eval	2
Dump Interval	0
Dumps Saved	0
Options	Acle
Random Seed	3237849206
Maximum Bias	0.99
Max Convergence	160
Conv Threshold	0.8
DPE Time Constant	0
Sigma Scaling	2.0
Init Weight Range	1.0
Max Weight Range	100.0
Crossover Prob	0.15
Delta1Range*	2.0
Delta2Range*	0.1
Continuous Output	0
Cont. Tolerance	0.0
Loopback Weight	0.0
Normalise	0
Output Layer	S
Early Stop	0.0

* (0.2, 0.2) best for re-evaluation (see table 3)

Table 21 4-2-4 encoder network input file.

B.3 The Digital to Analogue Converter Network

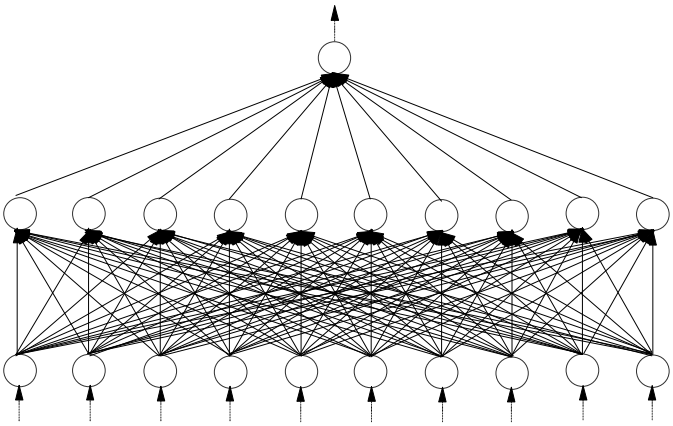


Figure 24 The digital to analogue converter network.

The digital to analogue converter network shown in figure 24 was trained with the training input vectors shown in table 22 to produce the expected output shown.

Input Vector	Expected Output
1000000000	0.1
0100000000	0.2
0010000000	0.3
0001000000	0.4
0000100000	0.5
0000010000	0.6
0000001000	0.7
0000000100	0.8
0000000010	0.9
0000000001	1.0

Table 22 Digital to analogue converter network expected operation.

The genetic algorithm input file for the digital to analogue converter network:

Parameter	Value
Experiments	50
Total Trials	50000
Population Size	1000
Structure Length	30
Crossover Rate	1.0
Mutation Rate	0.005
Generation Gap	0.7
Scaling Window	-1
Report Interval	1
Structures Saved	0
Max Gens w/o Eval	2
Dump Interval	0
Dumps Saved	0
Options	Acle
Random Seed	3237849206
Maximum Bias	0.99
Max Convergence	750
Conv Threshold	0.8
DPE Time Constant	0
Sigma Scaling	0.0
Init Weight Range	1.0
Max Weight Range	50.0
Crossover Prob	0.2
Delta1Range	0.04
Delta2Range	0.02
Continuous Output	1
Cont. Tolerance	-0.03
Loopback Weight	0.0
Normalise	0
Output Layer	S
Early Stop	0.0

Table 23 Digital to analogue converter network input file.

B.4 The Sequence Generator Network

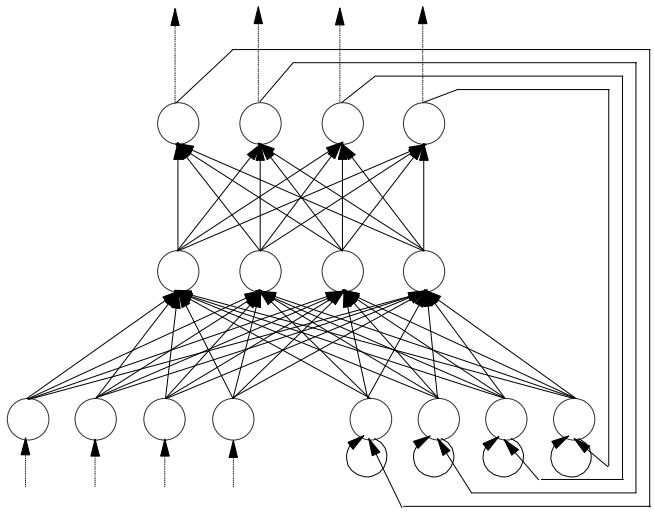


Figure 25 The sequence generator network.

The sequence generator network shown in figure 25 was trained with the training input vectors shown in table 24 to produce the expected output shown.

Input Vector	Expected Output
1010	1000
1010	0100
1010	0010
1010	0001
0101	0001
0101	0010
0101	0100
0101	1000

Table 24 Sequence generator network expected operation.

The genetic algorithm input file for the sequence generator network:

Parameter	Value
Experiments	50
Total Trials	100000
Population Size	500
Structure Length	30
Crossover Rate	1.0
Mutation Rate	0.005
Generation Gap	0.9
Scaling Window	-1
Report Interval	1
Structures Saved	0
Max Gens w/o Eval	2
Dump Interval	0
Dumps Saved	0
Options	Acle
Random Seed	3237849206
Maximum Bias	0.99
Max Convergence	300
Conv Threshold	0.8
DPE Time Constant	0
Sigma Scaling	4.0
Init Weight Range	1.0
Max Weight Range	100.0
Crossover Prob	0.2
Delta1Range	0.05
Delta2Range	0.05
Continuous Output	0
Cont. Tolerance	0.0
Loopback Weight	0.5
Normalise	0
Output Layer	S
Early Stop	0.0

Table 25 Sequence generator network input file.

B.5 The Sine Network

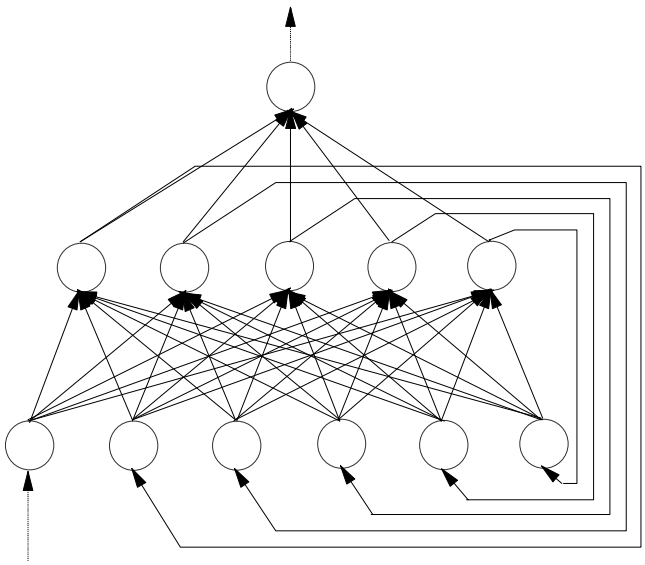


Figure 26 The sine network.

Values from the sine curve were presented one at a time as input to the sine network shown in figure 26, which was trained to predict the next value in the sequence. The training, validation, and testing data for this network were generated specifically for this thesis by selecting *100* values from the sine curve (for each data set) between *0* radians and 2π radians. Since this data can be easily reproduced it is not presented here.

The genetic algorithm input file for the sine network:

Parameter	Value
Experiments	10
Total Trials	7500
Population Size	100
Structure Length	30
Crossover Rate	1.0
Mutation Rate	0.005
Generation Gap	0.9
Scaling Window	-1
Report Interval	1
Structures Saved	0
Max Gens w/o Eval	2
Dump Interval	0
Dumps Saved	0
Options	Acle
Random Seed	3237849206
Maximum Bias	0.99
Max Convergence	85
Conv Threshold	0.8
DPE Time Constant	0
Sigma Scaling	4.0
Init Weight Range	1.0
Max Weight Range	100.0
Crossover Prob	*(varied)*
Delta1Range	0.003
Delta2Range	0.003
Continuous Output	1
Cont. Tolerance	0.1
Loopback Weight	0.0
Normalise	0
Output Layer	L
Early Stop	0.0

Table 26 Sine network input file.

B.6 The Sunspot Simple Recurrent Network

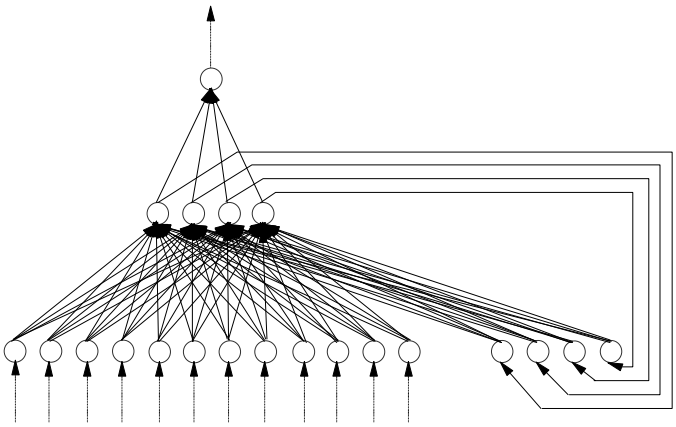


Figure 27 The sunspot simple recurrent network.

The sunspot simple recurrent network shown in figure 27 was presented as input a moving window of twelve values from the recorded sunspot data, and trained to predict the next value in the sequence. The training, validation, and testing data for this network are from [Tong, 1991]. The complete sunspot data from *1700* to *1979* is presented in Appendix C.

The genetic algorithm input file for the sunspot simple recurrent network:

Parameter	Value
Experiments*	5
Total Trials	50000
Population Size	1000
Structure Length	30
Crossover Rate	1.0
Mutation Rate	0.005
Generation Gap	0.9
Scaling Window	-1
Report Interval	1
Structures Saved	0
Max Gens w/o Eval	2
Dump Interval	0
Dumps Saved	0
Options	Acle
Random Seed	3237849206
Maximum Bias	0.99
Max Convergence	700
Conv Threshold	0.8
DPE Time Constant	0
Sigma Scaling	4.0
Init Weight Range	1.0
Max Weight Range	100.0
Crossover Prob	*(varied)*
Delta1Range	0.01
Delta2Range	0.01
Continuous Output	1
Cont. Tolerance	0.1
Loopback Weight	0.0
Normalise	1
Output Layer	S
Early Stop**	0.0

* 10 for the early stopping tests
** 15.0 for the early stopping tests

Table 27 Sunspot SRN input file.

B.7 The Sunspot Real Time Recurrent Network

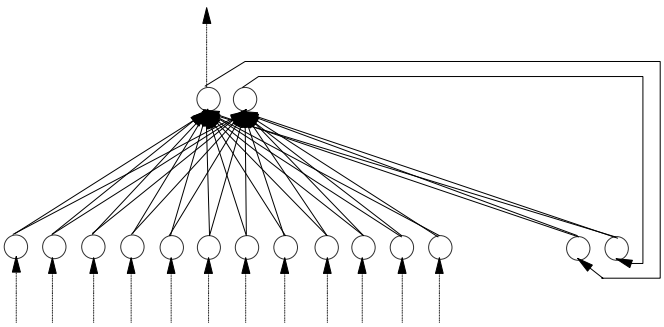


Figure 28 The sunspot real time recurrent network.

The sunspot real time recurrent network shown in figure 28 was presented as input a moving window of twelve values from the recorded sunspot data, and trained to predict the next value in the sequence. The training, validation, and testing data for this network are from [Tong, 1991]. The complete sunspot data from *1700* to *1979* is presented in Appendix C.

The genetic algorithm input file for the sunspot real time recurrent network:

Parameter	Value
Experiments*	5
Total Trials	50000
Population Size	1000
Structure Length	30
Crossover Rate	1.0
Mutation Rate	0.005
Generation Gap	0.9
Scaling Window	-1
Report Interval	1
Structures Saved	0
Max Gens w/o Eval	2
Dump Interval	0
Dumps Saved	0
Options	Acle
Random Seed	3237849206
Maximum Bias	0.99
Max Convergence	700
Conv Threshold	0.8
DPE Time Constant	0
Sigma Scaling	4.0
Init Weight Range	1.0
Max Weight Range	100.0
Crossover Prob	*(varied)*
Delta1Range	0.01
Delta2Range	0.01
Continuous Output	1
Cont. Tolerance	0.1
Loopback Weight	0.0
Normalise	1
Output Layer	S
Early Stop**	0.0

* 10 for the early stopping tests
** 15.0 for the early stopping tests

Table 28 Sunspot RTRN input file.

Appendix C
Sunspot Data

Year	Value	Year	Value	Year	Value	Year	Value	Year	Value	Year	Value
1700	5.0	**1747**	40.0	**1794**	41.0	**1841**	36.7	**1888**	6.8	**1935**	36.1
1701	11.0	**1748**	60.0	**1795**	21.3	**1842**	24.2	**1889**	6.3	**1936**	79.7
1702	16.0	**1749**	80.9	**1796**	16.0	**1843**	10.7	**1890**	7.1	**1937**	114.4
1703	23.0	**1750**	83.4	**1797**	6.4	**1844**	15.0	**1891**	35.6	**1938**	109.6
1704	36.0	**1751**	47.7	**1798**	4.1	**1845**	40.1	**1892**	73.0	**1939**	88.8
1705	58.0	**1752**	47.8	**1799**	6.8	**1846**	61.5	**1893**	85.1	**1940**	67.8
1706	29.0	**1753**	30.7	**1800**	14.5	**1847**	98.5	**1894**	78.0	**1941**	47.5
1707	20.0	**1754**	12.2	**1801**	34.0	**1848**	124.7	**1895**	64.0	**1942**	30.6
1708	10.0	**1755**	9.6	**1802**	45.0	**1849**	96.3	**1896**	41.8	**1943**	16.3
1709	8.0	**1756**	10.2	**1803**	43.1	**1850**	66.6	**1897**	26.2	**1944**	9.6
1710	3.0	**1757**	32.4	**1804**	47.5	**1851**	64.5	**1898**	26.7	**1945**	33.2
1711	0.0	**1758**	47.6	**1805**	42.2	**1852**	54.1	**1899**	12.1	**1946**	92.6
1712	0.0	**1759**	54.0	**1806**	28.1	**1853**	39.0	**1900**	9.5	**1947**	151.6
1713	2.0	**1760**	62.9	**1807**	10.1	**1854**	20.6	**1901**	2.7	**1948**	136.3
1714	11.0	**1761**	85.9	**1808**	8.1	**1855**	6.7	**1902**	5.0	**1949**	134.7
1715	27.0	**1762**	61.2	**1809**	2.5	**1856**	4.3	**1903**	24.4	**1950**	83.9
1716	47.0	**1763**	45.1	**1810**	0.0	**1857**	22.7	**1904**	42.0	**1951**	69.4
1717	63.0	**1764**	36.4	**1811**	1.4	**1858**	54.8	**1905**	63.5	**1952**	31.5
1718	60.0	**1765**	20.9	**1812**	5.0	**1859**	93.8	**1906**	53.8	**1953**	13.9
1719	39.0	**1766**	11.4	**1813**	12.2	**1860**	95.8	**1907**	62.0	**1954**	4.4
1720	28.0	**1767**	37.8	**1814**	13.9	**1861**	77.2	**1908**	48.5	**1955**	38.0
1721	26.0	**1768**	69.8	**1815**	35.4	**1862**	59.1	**1909**	43.9	**1956**	141.7
1722	22.0	**1769**	106.1	**1816**	45.8	**1863**	44.0	**1910**	18.6	**1957**	190.2
1723	11.0	**1770**	100.8	**1817**	41.1	**1864**	47.0	**1911**	5.7	**1958**	184.8
1724	21.0	**1771**	81.6	**1818**	30.1	**1865**	30.5	**1912**	3.6	**1959**	159.0
1725	40.0	**1772**	66.5	**1819**	23.9	**1866**	16.3	**1913**	1.4	**1960**	112.3
1726	78.0	**1773**	34.8	**1820**	15.6	**1867**	7.3	**1914**	9.6	**1961**	53.9
1727	122.0	**1774**	30.6	**1821**	6.6	**1868**	37.6	**1915**	47.4	**1962**	37.5
1728	103.0	**1775**	7.0	**1822**	4.0	**1869**	74.0	**1916**	57.1	**1963**	27.9
1729	73.0	**1776**	19.8	**1823**	1.8	**1870**	139.0	**1917**	103.9	**1964**	10.2
1730	47.0	**1777**	92.5	**1824**	8.5	**1871**	111.2	**1918**	80.6	**1965**	15.1
1731	35.0	**1778**	154.4	**1825**	16.6	**1872**	101.6	**1919**	63.6	**1966**	47.0
1732	11.0	**1779**	125.9	**1826**	36.3	**1873**	66.2	**1920**	37.6	**1967**	93.8
1733	5.0	**1780**	84.8	**1827**	49.6	**1874**	44.7	**1921**	26.1	**1968**	105.9
1734	16.0	**1781**	68.1	**1828**	64.2	**1875**	17.0	**1922**	14.2	**1969**	105.5
1735	34.0	**1782**	38.5	**1829**	67.0	**1876**	11.3	**1923**	5.8	**1970**	104.5
1736	70.0	**1783**	22.8	**1830**	70.9	**1877**	12.4	**1924**	16.7	**1971**	66.6
1737	81.0	**1784**	10.2	**1831**	47.8	**1878**	3.4	**1925**	44.3	**1972**	68.9
1738	111.0	**1785**	24.1	**1832**	27.5	**1879**	6.0	**1926**	63.9	**1973**	38.0
1739	101.0	**1786**	82.9	**1833**	8.5	**1880**	32.3	**1927**	69.0	**1974**	34.5
1740	73.0	**1787**	132.0	**1834**	13.2	**1881**	54.3	**1928**	77.8	**1975**	15.5
1741	40.0	**1788**	130.9	**1835**	56.9	**1882**	59.7	**1929**	64.9	**1976**	12.6
1742	20.0	**1789**	118.1	**1836**	121.5	**1883**	63.7	**1930**	35.7	**1977**	27.5
1743	16.0	**1790**	89.9	**1837**	138.3	**1884**	63.5	**1931**	21.2	**1978**	92.5
1744	5.0	**1791**	66.6	**1838**	103.2	**1885**	52.2	**1932**	11.1	**1979**	155.4
1745	11.0	**1792**	60.0	**1839**	85.7	**1886**	25.4	**1933**	5.7		
1746	22.0	**1793**	46.9	**1840**	64.6	**1887**	13.1	**1934**	8.7		

Table 29 Sunspot data from 1700 to 1979.