A Neuro-Evolutionary Approach to Time Series Forecasting

A Project Report submitted as part of the
SURGE 2006 Program

by
Sidhant Dash
(Y4429)

Department of Mathematics and Statistics
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
KANPUR 208016, INDIA
JULY 2006

Abstract

Time Series Forecasting is an extremely important problem that has received a lot of attention from many different quarters of science and engineering because of its intrinsic difficulties and practical applications. We present a Neuro-Evolutionary approach to time series forecasting, wherein we evolve neural network models to do time series prediction using an advanced evolutionary computation algorithm, called Neuro-Evolution using Augmenting Topologies. Changes in the original algorithm are proposed and to reach the high level of precision desired by time series forecasting, a novel ensembling method is also proposed and implemented.

The algorithms ability to evolve networks for a chaotic time series prediction task is tested by applying it to a benchmark series, the Mackey-Glass Time Series, and finally to test the proposed ensembling technique, the system is applied to predicting a complex currency exchange rate time series, that of the Yen-$ daily exchange rates.

The results suggest that the system is robust, and when combined with the proposed ensembling technique, it is able to beat the predictions of the Random Walk model in the currency exchange rate prediction task. In the Mackey Glass test, the system successfully demonstrates its ability to evolve minimal structures to solve complex problems and multiple runs on the task confirm its ability to create diversity in terms of the architectures of the neural networks it produces.

Acknowledgements

This project has been successful because of the encouragement and guidance of my mentor Dr Amit Mitra, Department of Mathematics and Statistics, and my co-mentor Dr Neeraj Misra, Department of Mathematics and Statistics, my parents, whose moral support has been invaluable throughout the SURGE program.

Sidhant Dash

Certificate

It is certified that the work contained in this report titled "*A Neuro-Evolutionary Approach to Time Series Forecasting"* is the original work done by *Sidhant Dash (Y4429),* and has been carried out under our supervision.

Dr Amit Mitra
Mentor
Department of Mathematics and Statistics
Indian Institute of Technology, Kanpur

Dr Neeraj Misra
Co-Mentor
Department of Mathematics and Statistics
Indian Institute of Technology, Kanpur

12th July, 2006

Table of Contents

List of Tables

List of Figures

List of Symbols

Symbol	*Description*
$\xi(n)$	Total Error Energy in the Back Propagation Algorithm
ξ_{av}	The *average squared error Energy* in Back Propagation
η	Learning rate parameter
$\delta_j(n)$	The Local gradient in error calculations in Back Propagation, for *j*th neuron at nth iteration
$\beta(n)$	Scaling factor in the Conjugate Gradient method
$\Phi_m(x)$	Output of an Ensemble having *m* generalizers, given an input vector *x*.
$\theta(t)$	The set of RTRL layer nodes for which a specified target for their output exists
w_{ij}	Weight of the connection from node *j* to node *i*

Chapter 1
Introduction

Building a forecasting model for exchange rates has always been a challenging area of research to applied econometricians and statisticians. It has been observed that the forecasts arising out of several important structural models based on monetary and asset theories of exchange rate determination perform no better than (in terms of out-of-sample forecasting ability) ones generated by the simplest of all models, i.e. the simple random walk model [Meese and Rogoff, 1983]. These findings have also been confirmed for a number of key exchange rates over the period 1976-1984, i.e. the Bretton Woods era [Boothe and Glassman, 1987]. Other studies, like [Alexander and Thomas, 1987] have further showed that these econometric models are outperformed by the random walk model even when time-varying parameters are incorporated into the models. Although these results are supportive of the efficient-market hypothesis, researchers have continued to explore various alternate techniques for modelling of exchange rates.

These empirical studies have mainly relied on linear time series techniques and hence it is not unreasonable to believe that the linear unpredictability of exchange rates may be due to limitations of linear models. [Meese and Rose, 1991] also show that exchange rates exhibit significant non-linearities. [Chinn, 1991] also provides some support for non-linear exchange rate models by using the alternating conditional expectations model.

Use of neural network based models is an alternative option available to researchers for capturing the underlying non-linearity in the exchange rate series. There are several features of the artificial neural network (ANN) based models that make them attractive as a forecasting tool. First, as opposed to the traditional model based methods, ANN based models are data driven and self-adaptive. Second, ANNs are universal function approximators. It has been shown that a network can approximate any continuous function to any desired accuracy [Hornik, 1991]. Finally ANNs are non linear models. Under ANN, it is not necessary to specify the functional relationship between input and output variables, and this gives it a distinct advantage over other non-linear models. In recent years, ANN had been used as a successful tool for forecasting. [Zhang *et al*, 1998] provide a comprehensive review of the current status of research in this area.

Exchange rate forecasting using artificial neural networks (ANN) provides evidence that they are significantly better than existing statistical methods in terms of out-of-sample forecasting ability. [Refenes, 1996] observes that with hourly exchange rates data, ANN model performs much better than exponential smoothing and auto-regressive integrated moving average (ARIMA) models. For daily frequency exchange rate data, [Weigend *et al*, 1992] proved that ANN is significantly better than the random walk model. With weekly exchange data [Hann and Steurer, 1996] showed that ANN outperforms the linear models. [Lisi and Schiavo, 1999] performed a detailed comparision of neural network models and chaotic models for predicting monthly exchange rates. They found that neural networks compare favourably with chaotic models and both perform significantly better than random walk models.

The success of ANN-based methods for exchange rate forecasting at various frequencies is thus a well-established fact. It is however, well known [Refenes, 1996] that a solution obtained through standard neural network algorithms like back-propagation suffers from serious drawbacks. These include the possibility of convergence to a local minima and a search for optimum network architecture through experimentation. Since there are no *a priori* realistic estimates for parameters of the network architecture, network design heavily depends on manual experimentation and fine-tuning of the learning parameters. Thus although ANN models for exchange rates report superior performance there is still room for further improvement.

Here we explore a neuro evolutionary approach to predicting a noisy financial time series, using a method called NeuroEvolution using Augmented Topologies (NEAT; [Stanley and Miikkulainen. 2002]). In NEAT, the topology as well as the weights of the neural networks are varied simultaneously to prevent their being stuck in local minima. Variations in the original neat algorithm are implemented to make it suitable for the problem at hand. Networks produced by NEAT are further fine-tuned using a conjugate gradient based back-propagation method to further improve their performance. An ensemble of the resulting networks is then seen to be able to beat the Random Walk model’s predictions on the same data set.

Chapter 2

Neuro Evolution

2.1 Definition

Neuroevolution (NE), the artificial evolution of neural networks using genetic algorithms, has shown great promise in complex reinforcement learning tasks [Gomez and Miikkulainen, 1999]. Neuroevolution searches through the space of behaviours for a network that performs well at a given task. This approach to solving complex control problems represents an alternative to statistical techniques that attempt to estimate the utility of particular actions in particular states of the world [Kaelbing *et al*, 1996]. Because NE searches for a behaviour instead of a value function, it is effective in problems with continuous and high dimensional state spaces. In addition memory is easily represented through recurrent connections in neural networks.

In traditional NE approaches, a topology is chosen for the evolving networks before the experiment begins. Usually, the network topology is a single hidden layer of neurons, with each hidden neuron connected to every network input and every network output. Evolution then proceeds to search the space of the connection weights of the networks by allowing high-performing networks to reproduce. The weight space is explored through the cross over of network weight vectors and through the mutation of of single networks' weights. Thus the goal of fixed-topology NE is to optimize the connection weights that determine the functionality of a network.

However, connection weights are not the only aspect of neural networks that contribute to their behaviour. The topology, or structure, of neural networks is another important that affects their functionality. Structure has been shown to be of importance in supervised training [Chen *et al*, 1993]. A number of studies in the recent past have tried to evolve network topologies as well as weights simultaneously [Angeline *et al*, 1993, Branke, 1995, Gruau *et al*, 1996, Yao, 1999].

2.2 Why Evolve Topology

It is a well known result that a fully connected network can in principle approximate any continuous function [Cybenko, 1989]. So the question that arises is why should we invest valuable time and resources to search through the topology space?

[Gruau *et al*, 1996] claimed that evolving structure saves the time wasted by humans trying to decide the topology of networks

For a particular NE problem, all fixed topology networks use a fully connected hidden layer, but the number of hidden layer neurons can only be decided by a trial and error process. [Zhang and Muhlenbein, 1993] argue that network complexity can affect the speed and accuracy of learning. This claim is certainly true for back-propagation algorithm, but it is not known if the claim is valid if the weights are optimized by evolution and not back-propagation. Additionally, for parameters like the number of hidden layer neurons, the number of input neurons and the connectivity, the absence of any *a priori* information, makes the choice of these parameters rather subjective, and so the success of the neural network now depends on the experience of the the experimenter. As a result of a subjective choice of these structural parameters of the ANN, there is always a danger of the network getting stuck in local minima. So it becomes important that the network architecture is also optimized, along with the connection weights.

If done right, evolving structure along with the connection weights can significantly enhance the performance of NE.

2.3 Past work in Neuro Evolution

Many different systems have been developed over the past decade that evolve topology and weights in neural networks simultaneously [Angeline et al, 1993; Braun and Weisbrod, 1993; Gruau et al, 1996; Krishnan and Ciesielski, 1994; Yao and Liu, 1996]. These methods cover a large variety of methods regarding how topology and weights in a neural network could be simultaneously evolved. In this sub-section some of the major drawbacks of past neuro-evolutionary approaches are highlighted, and possible solutions are discussed. The neuro evolution approach used for this project is NeuroEvolution using Augmented Topologies (NEAT; [Stanley and Miikkulainen. 2002]).

2.3.1 Protecting Innovation with Speciation

In most *Topology and Weight Evolving Artificial Neural Networks* (TWEANNs), innovations take place by adding new structure to networks through mutation, or sometimes deleting existing structure [Yao and Liu, 1996]. A new node or connection addition will almost certainly produce an instantaneous decrease in the fitness of the network. The network needs some time before the advantages of the mutation start showing some results, like for example the connection is optimized. Some generations are

required for such a change to come about. However, the initial loss of fitness as a result of an architectural mutation makes it difficult for the network to survive in the population. It needs to survive long enough for the mutation to be optimized. Thus it is necessary to somehow protect networks with structural innovations so they have a chance to make use of their new structure.

In nature different organisms belong to different species, and need to compete with other organisms of the same species for their survival. So if networks with structural innovations could be somehow isolated into different species', they could get the time necessary to optimize their structures.

Speciation, also known as *niching*, has been studied in GAs, but it is not usually applied to neuroevolution. Speciation is most commonly applied to multimodal function optimization [Mahfoud, 1995], where a function has multiple optima, and a GA with several species is used to find those optima.

Speciation requires a compatibility function to tell whether two genomes should be in the same species or not. TWEANNs may not have implemented speciation because it is rather difficult to form a compatibility function when the population has individuals of many different topologies. NEAT however solves this problem by using historical information about genes. The population developed for this project had three *types* of networks, namely feed-forward networks (Multi Layer Perceptrons, MLPs), Elman networks (ELMs) and general recurrent networks (RNNs; [Williams and Zipser, 1989]). For the ELM and MLP networks speciation was carried out solely on the basis of the number of input and hidden nodes, and for the RNNs, a compatibility function based on historical information of the genes was used.

2.3.2 Initial Population and Topological Innovation

In many TWEANN systems, the initial population is a collection of random topologies. Such a population ensures topological diversity from the start. However, for the purposes this was not a desirable option.

The goal of our experiment was to use a neuro-evolutionary approach to evolve networks for a supervised learning task, namely time series forecasting. The neuro-evolutionary approach chosen for the task, NEAT, does not use any gradient information, and weight changes are solely through random mutations. Such an approach may not be able to produce networks that could perform a supervised task with the desired precision. So, the networks evolved by NEAT were to be put through a gradient-descent based

training process for further improvement and fine-tuning. Additionally, there is the maxim that gradient information when available should be used.

It is known that training RNNs with gradient descent methods is practical only with smaller networks, as otherwise with longer temporal dependencies , the gradient vanishes as the error signal is propagated back through time so that network weights are never adjusted correctly to account for events in the past [Hochreiter *et al*, 2001].

So it becomes necessary for the networks to be minimal in structure. One way to force minimal topologies is to incorporate network size into the fitness function. Some TWEANNs actually do this [Zhang and Muhlenbein, 1993]. Such setups favour smaller networks over larger ones, which at times may not be desirable as the optimal network size to solve a particular problem may not be known. Additionally as to how much a network is to be penalized when it crosses a certain size threshold is also not known.

NEAT solves this problem by starting out minimally and adding structure gradually to benefit the solution. In the initial RNN population, the individuals have no hidden nodes, and in the ELM and MLP individuals, there are only 2-3 hidden nodes (just to maintain connectivity in the network).

Chapter 3
NEAT

The NEAT method, as used by us, is described in detail in this section.

3.1 The '*types*' in the Initial Population

As was mentioned in the previous section, NEAT protects innovation in its population through speciation, and this doesn't necessitate starting out with topological diversity. In this project NEAT was used to evolve networks for a time-series prediction task.

It has been shown that multi-layer feedforward networks with one hidden layer of neurons can be used to approximate any Borel-measurable functions to any desired level of accuracy [Lapedes and Farber, 1988; Kosko, 1992]. Thus feed-forward networks having one layer of hidden weights, are added to the initial population, and are hence-forth referred to as the *type-1*(or MLP) neural networks. The forecasting abilities of such networks in time series forecasting tasks are well established.

Recent studies have also established the usefulness of RNNs in time series forecasting tasks. Two different kinds of RNNs are used in the initial population of NEAT for the purpose of this project, namely Elman Networks (henceforth referred to as *type-2*(or ELM) neural networks [Elman, 1990] and completely recurrent networks, where any node can be connected to any other node in the network, henceforth referred to in this report as *type-3*(or RTRL) networks [Williams and Zipser, 1989; Chang and Mak, 1998].

So, NEAT starts out with certain diversity in the initial population, with three different *types* of networks in the initial population. The mutations that are part of the NEAT methodology are also slightly different when it comes to the different *types*, as shall be seen later.

3.2 Genetic Encoding in the *type-3* Networks

To develop a practically feasible cross-over operation for the RTRL networks, NEAT follows a genetic encoding scheme that makes it easier for corresponding genes to be lined up during crossover. Each genetic encoded network has two lists : (1) *connectiongenes*, (2) *nodegenes*.

For every single connection between two nodes in a network, there exists a corresponding entry in the *connectiongenes* list. Each

entry in the *connectiongenes* list species the in-node, out-node, the connection weight associated with the link, an enable bit that specifies whether or not the connection is active, and finally an innovation number, which gives information about the historical origin of the genes, and is used for the line-up during crossover.

In the diverse population of the RTRL networks, it is the historical origin of a gene that tells when one gene matches up with another. Two genes, in two different RTRL networks, having the same historical origin, must match up, inspite of their having different connection weights in the two networks. Tracking the historical origins of the genes is accomplished in NEAT by using the idea of a *global innovation number*. The moment a new structural innovation in the form of a new connection appears in an individual *type-3* network in the population, it is given a new innovation number, and this new structure, along with its innovation number is stored in an *innovation database*.

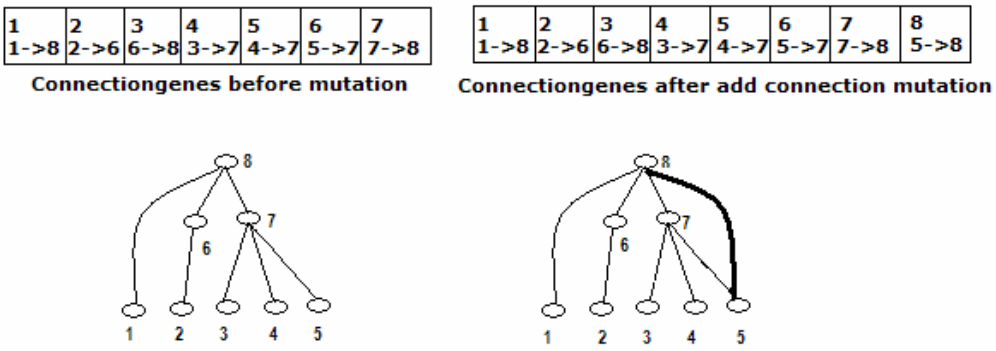

Figure 3.1: Add connection mutation in RTRL networks

The above figure shows the add connection mutation when applied to RTRL networks. The top number in the connectiongenes list is the innovation number of that gene. As new connections (genes) are evolved, they are given higher innovation numbers, and subsequently the connection is also saved in the innovation database. In case a connection is to be added to a network that already exists in the innovation database, i.e. the particular mutation has already occurred previously, then the corresponding gene is retrieved from the database and it is added to the connectiongenes matrix of the network with the innovation number of the original mutation as stored in the innovation database.

The other kind of mutation that is used in NEAT is the add node mutation. For the RTRL networks, the version of NEAT implemented has two different mutations:

(1) Add Hidden Node

(2) Add Input Node

NEAT originally had only the 'Add Hidden Node' mutation [Stanley and Miikkulainen, 2002]. Later on the makers also added the input node mutation in a second version of NEAT (FS-NEAT [Whiteson *et al.* 2005]. In FS-NEAT, the initial population consists of pools of inputs and outputs, with almost no connections, and as in NEAT, links and hidden nodes are added through mutations, and only those additions that aid performance are likely to survive. The useful inputs that get connected through the mutations are going to stay, and redundant inputs, if any, will be gradually phased out by the evolutionary process. Thus feature selection occurs implicitly in FS-NEAT.

The time series prediction task that we were interested in was different in the sense that there were no predefined inputs. So, it was not possible to build a pool of inputs for FS-NEAT to begin with. We however needed to have the input node mutation because the number of lags (past values) of the input variable is an important parameter in the prediction process, and we hoped to discover this through the evolutionary process. So the initial population consisted of a bare minimum of input nodes, and the 'Add Input Node' mutation proceeded by directly adding a new input node and connecting it to only the output node. This is similar to the add input node mutation done in an older version of FS-NEAT, with the only difference that in the initial population instead of a bias and a random input connected to the output, we started out with a fixed number of input nodes already connected to the output, and more input nodes added as evolution proceeded.

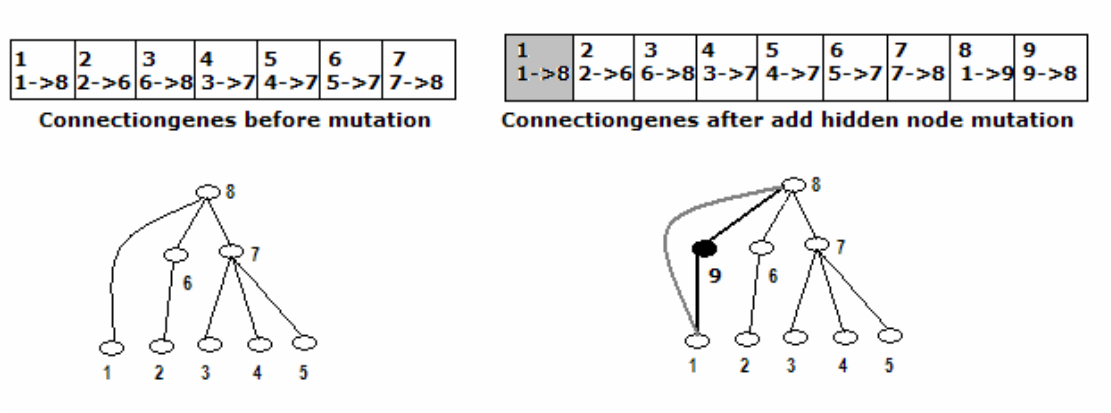

Figure 3.2: Add hidden node mutation in the RTRL networks.

The gene connecting 1->8 is disabled (greyed in the connectiongenes, Figure 3.2) and new genes added for the new hidden node. In this mutation, the new connection going into the newly added node 9 gets a weight of 1, and the new connection

going from node 9->8 receives the same connection weight as the old connection (1->8) which has now been disabled. This is done in NEAT to minimize the initial effect of the mutation.

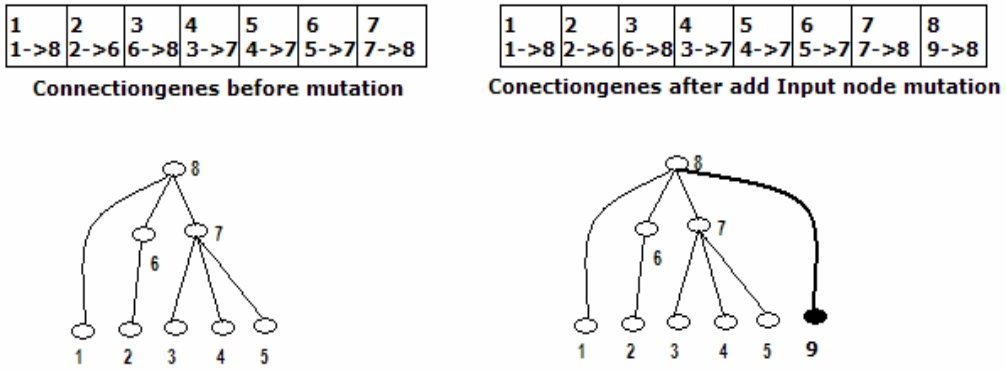

Figure 3.3: The Add Input Node mutation for RTRL networks.

The new node that is added is connected directly only to the output node, and not to any of the existing hidden nodes. The connectiongenes list has been appropriately updated. Input node addition does not require disabling any existing genes as in the hidden node addition. The new connection is given a random weight between -0.5 and 0.5.

Mutation in NEAT also perturbs the weights of various connections in the network randomly. This happens as in any NE system.

Through mutations the genomes of the RTRL networks will gradually get larger. NEAT imposes no bounds on the size of the genomes, but still manages to cross over differently sized genomes by using the historical information stored in the genes in the form of the *innovation numbers.* This is explained in detail in the following section.

3.3 Tracking Genes through Historical Markings

The idea of an innovation number as a marker of the historical origins of a gene was introduced in the previous section. In our experiments, the innovation database was maintained for the entire period of evolution, and this prevented duplication of any structural innovations. So even if a particular innovation occurred a second time during evolution, it still will receive the same innovation number. Thus the innovation number uniquely determines structural innovation throughout the evolutionary process.

Using innovation numbers as historical markers, NEAT develops a powerful method to line-up genes of the *type-3* (RTRL) networks during crossover.

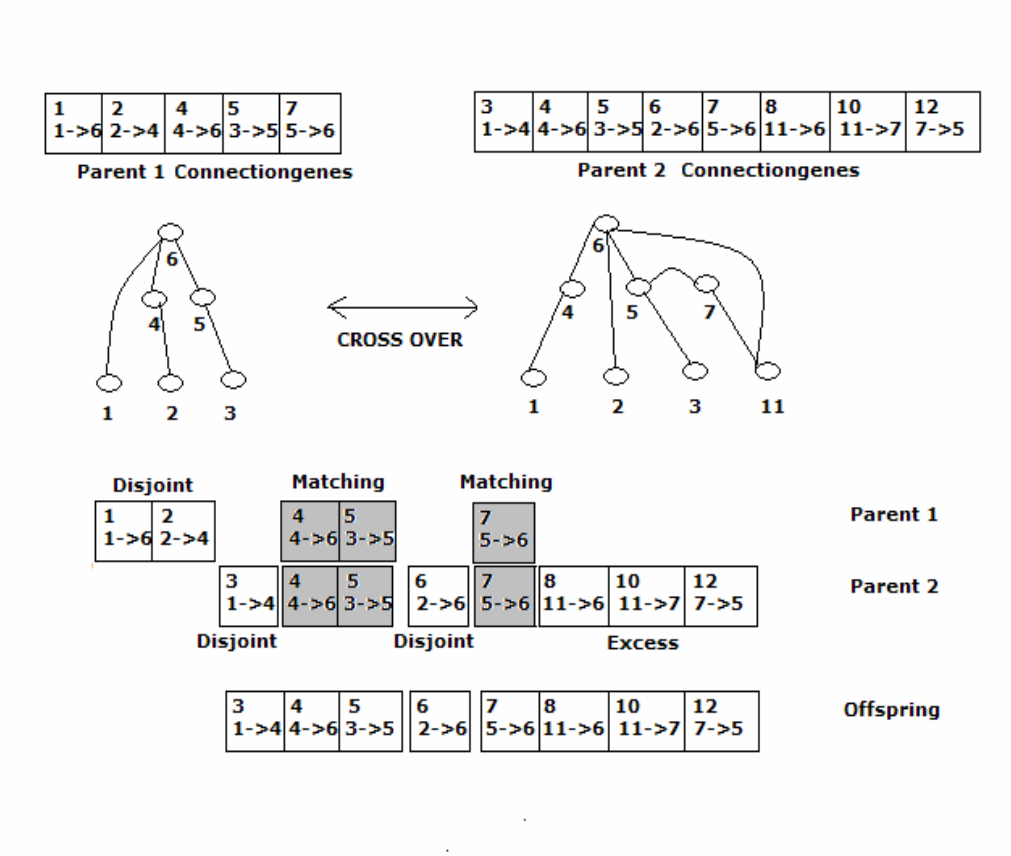

Figure 3.4: Matching up of genomes in RTRL networks

When crossing over, the genes in both the connectiongenes with the same innovation number are lined up. Genes that do not match are either *disjoint* or *excess*, depending on whether they lie within or beyond the other parent's innovation numbers. The offspring gets the matching genes in a random fashion, i.e., from either of the parents. In case of the disjoint and excess genes, the offspring inherits the disjoint or excess genes only from the parent with a larger fitness. In figure 3.4, parent 2 is the more fit parent, and so the offspring inherits disjoint or excess genes only from parent 2.

Thus using historical information of the genes, NEAT is able to mate networks with diverse topology without using excessive computation power. In the next section we shall see the genetic encoding of the two other *types* of networks in the population, namely *type-1* and *type-2.*

3.4 Genetic Encoding in the *type-1* and *type-2* networks

The initial population in the version of NEAT implemented for the purposes of this project had three *types* in the initial population, as has already been mentioned in section 3.1. The crossover and mutation operator for the *type-3* networks has already been described. In this section, we shall be looking at the mutation operator of the *type-1 and type-2* networks. The genetic encoding for these types also comprises of a connectiongenes and nodegenes list with entries analogous to the RTRL networks.

3.4.1 Crossover

In the case of *type-1*(MLP) and *type-2*(ELM) networks, we do not employ a cross-over operator. No innovation database is maintained for these types of networks, as at no point in the evolution do we mate them. In the crossover stage of NEAT, when we are dealing with *type-1* and *type-2* networks, the offspring simply inherits the nodes and connections (weights) from one of the parents. As inter-species mating is not used in our implementation of NEAT, the two parents have exactly the same structure in terms of the different kinds and numbers of nodes and the connectivity, and so it is immaterial as to which parent's nodes are inherited by the offspring. Both the parents are assumed to be of the same fitness, and the task of bringing about diversity and improvement is solely that of the various *mutation* operators of the *type-1* and *type-2* networks.

3.4.2 Mutation

Three different kinds of mutation operators work on these networks,

(a) Weight mutation
(b) Add input node mutation
(c) Add hidden node mutation

The Weight mutation is like in the RTRL networks, some of the weights in the network are perturbed according to a previously set parameter. In the 'Add Input Node' mutation, a new input node is added to the network, and it is completely interconnected to the layer above it, i.e. the hidden layer. The 'Add Input Node' mutation of the MLP/ELM networks causes the input node to be not connected to the output node, but rather it is connected to all the hidden nodes.

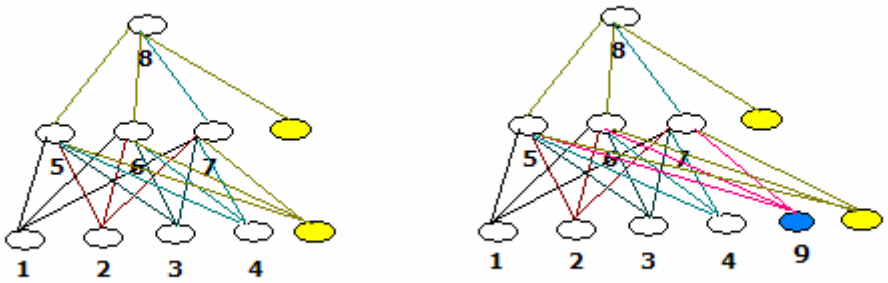

Figure 3.5: Add Input node mutation in the MLP networks

In the above figure, the left hand side shows the MLP network before the mutation. The yellow nodes are the bias nodes. On the right, the network after the 'Add Input Node' mutation is shown. Node number 9 is the newly added input node to the network, which has been completely interconnected with the hidden layer.

The 'Add Hidden Node' mutation of the MLP/ELM networks is analogous to the Add Input node mutation described in the above paragraph. In the MLP networks, a new hidden node is added and it is connected to all the output nodes. All the input nodes and the bias in the input layer also get connected to the new hidden node. The 'Add Hidden Node' mutation for the *type-2* (ELM) networks is slightly more complicated and is described in the following paragraph.

The ELM networks have context nodes that store the hidden layer activations of the previous time period. The number of such context units is equal to the number of the hidden layer nodes that the network has. So when a new hidden node is to be added, it has to be preceded by the addition of a new context node. So in a 'Add Hidden Node' mutation for the ELM networks, first a new context node is added to the network, which is completely interconnected with the hidden layer. This then is followed by the addition of the new hidden node, which is connected to all the output nodes, and also to all the input layer neurons, which include the input nodes, the input bias and the context units.

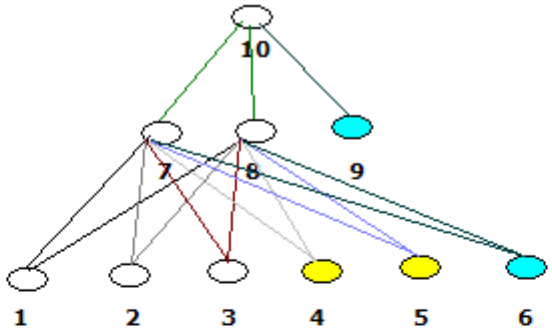

fig 3.6a ELM Network before the 'Add Hidden Node' mutation

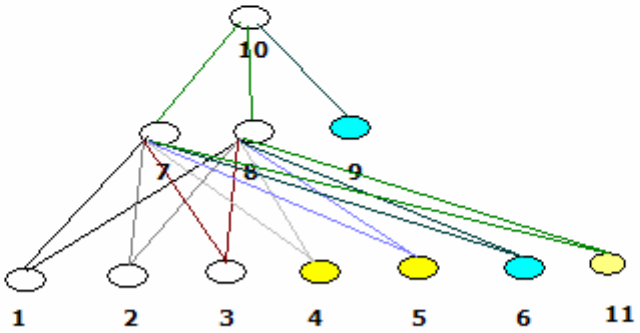

fig 3.6b ELM Network after the addition of the new *context* node

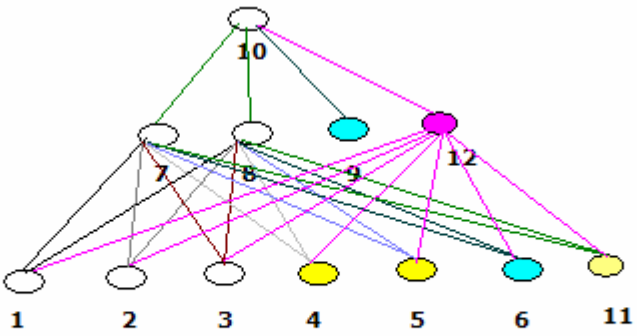

fig 3.6c ELM Network after the addition of the hidden node

Figure 3.6: 'Add Hidden Node' Mutation in ELM Networks

The 'Add Hidden Node' mutation is described in figure 3.6. In figure 3.6a we see the ELM node before the mutation. Nodes 6 and 9 are the bias nodes. In figure 3.6b, the first stage of the mutation is shown, a new *context* node (node 11) has been added, and it has also been connected to all the hidden nodes. Subsequently in figure 3.6c we see that a new hidden node (node 12) has been added to the network. It has been connected to the output node (node 10) and all the input layer nodes (nodes 1-6 and node 11) are also

connected to it. This completes the 'Add Hidden Node' mutation in the ELM network.

3.5 Speciation in NEAT

Speciating the population, as has been already mentioned allows organisms to compete primarily within their own niches instead of with the population at large. Thus structural innovations, are protected in their own niches where they have time to optimize their structure through competition within their niche. NEAT speciates the population by dividing them into distinct species, such that topologically similar networks are in the same species. In the version of NEAT implemented for the purposes of this project, the *type* information is also used for speciation.

3.5.1 Speciation in type-1 networks

In the speciation of the RTRL networks, NEAT uses an elegant solution based on the historical markings of the genes. The number of excess and disjoint genes between a pair of RTRL networks is a natural measure of their compatibility distance. Greater the number of disjoint genes in two genomes, the lesser their compatibility, and the lesser the evolutionary history they share. Thus for the RTRL (*type-3)* networks, the compatibility distance δ of different networks can be measured as a simple linear combination of the number of excess *E* and disjoint *D* genes, as well as the average weight differences of matching genes $\hat{W}$, including the disabled genes:

$$\delta = c1.E/N + c2.D/N + c3.\hat{W} \qquad (3.1)$$

The coefficients *c1, c2* and *c3* allow us to adjust the importance of the three factors, and the factor *N*, the number of genes in the larger genome, normalizes for genome size.

The distance measure δ allows us to speciate using a compatibility threshold δ_t. An ordered list of species is maintained. Each existing species is represented by a random genome inside the species from the previous generation. A given genome *g* in the current generation is placed in the first species in which *g* is compatible with the representative genome of that species, the compatibility being determined by the speciation threshold δ_t. This method of speciation ensures that the various species do not overlap. In case *g* happens to be incompatible with all the existing species, then a new species is created with *g* as the representative genome.

NEAT was used to evolve networks for a complex supervised learning task in our project. To create diversity in the *type-3*

networks, crossover was an important operator. For crossover, it was necessary to have at least two individuals of the species in question, inter-species mating was not used in our implementation of NEAT. An uncontrolled growth in the number of species could have created problems as then crossover would have become lesser effective, because it would have become more difficult to find sufficient individuals of the same species for crossover. Hence it was necessary that the number of species in NEAT be kept stable throughout the evolution.

To achieve this stability, an adaptive speciation threshold was used. A *speciation-change* parameter λ was defined. The speciation threshold δ_t was increased or decreased by a factor of the speciation-change parameter λ as the number of species exceeded or fell below a certain fixed value respectively.

3.5.2 Speciation in type-1 and type-2 networks

The reasons for speciation in *type-2* and *type-1* networks are analogous to those given for speciation of *type-3* networks, i.e., to protect topological innovation, and letting individuals compete only within their own niches.

Speciation in *type-2* and *type-1* networks is done solely on the basis of the number of input and hidden nodes they have. If we let *(I,H,t)* be the ordered triplet representing the number of *Input* neurons, *Hidden* neurons and the *type 't'* of a network, speciation causes every unique such triplet to be allotted to a separate species. Thus every single mutation (*Add Input node/Add Hidden node)* on the MLP/ELM networks creates a new species. The initial population in our implementation of NEAT, thus comprised of three species, each belonging to one of the three *types* of networks.

For evolution to be successful, we needed to keep the species numbers of the MLP/ELM networks stable too. The reasons for keeping the species numbers stable for these *types* of networks are however different from those given in the previous section for the RTRL networks. As has been mentioned earlier, we do not employ the same cross-over operator for the MLP/ELM networks, as we do for the RTRL networks. However, to keep the implementation simple, we did use the same internal representation for the nodes and the connections (*nodegenes* and *connectiongenes*). The *connectiongenes* matrices became extremely large as the sizes of the MLP/ELM networks increased, consequently slowing down the evolution process. Hence to prevent this slowing down, an additional parameter was added which controlled when a particular MLP/ELM network could undergo mutation. The value of this parameter (which was like the probability of a mutation occurring)

was increased or decreased respectively as the total number of existing species was lesser than or exceeded a pre-defined threshold value. This essentially acted as a check on the maximum number of species that could be evolved, and let evolution proceed meaningfully.

3.6 Fitness Sharing in NEAT

NEAT uses *explicit fitness sharing* [Goldberg and Richardson, 1987], where organisms in the same species must share the fitness of their niche. This puts a bar on the size of a species, it cannot grow too large even if a number of organisms in it perform well. Thus, any one species is not going to take over the entire population, which is crucial for the diversity in the population to be maintained and in order to evolve the best possible networks, or in other words, to reach the global *maxima*. The adjusted fitness f_i' for organism i is calculated according to its distance σ from every other organism j in the population

$$f_i' = f_i / \sum sh(\sigma(i,j)) \qquad (3.2)$$

The *sharing function* is set to 0 when i and j are in different species and is equal to 1 when i and j are in the same species. The denominator thus boils down to the number of individuals in the same species as organism i. Every species is assigned a particular number of offspring in proportion to the sum of the adjusted fitnesses f_i' of its member organisms. Species then reproduce by first copying into the next generation the best performing individual belonging to the species (*elitism)*, provided that the number of organisms in the species is above a certain threshold, then evolution proceeds by eliminating a preset percentage of the lowest performing individuals in the species. Finally, the entire population is replaced by the offspring of the remaining organisms in each species.

3.7 Minimizing Dimensionality in NEAT

Neat biases the search towards lower dimensional spaces by starting out with a population comprising of networks with the minimal possible structure. The *type-3* networks have no hidden nodes in the initial population, the inputs are connected only to the outputs, the *type-2* and *type-3* networks have a maximum of 2-3 hidden nodes solely to maintain connectivity between the three layers. In the *type-2* and *type-3* networks direct connectivity between the inputs and outputs isn't allowed, and so the initial networks started off with a minimal structure comprising of 2/3 hidden nodes .

New structure is added incrementally as structural mutations occur, and only those innovations survive and propagate that are found useful through fitness evaluations. Thus the structural elaborations in NEAT are always justified. Since the starting population is minimal, the dimensionality of the search space is minimized, and NEAT is always searching through fewer dimensions. Minimizing dimensionality gives NEAT a performance advantage compared to other approaches.

Chapter 4

Gradient Descent based Optimization

4.1 Introduction

The goal of this project was to evolve neural networks for time series forecasting, which is a supervised learning task that requires high precision. As has been mentioned earlier in this report, neuro-evolution alone may not be able to evolve such networks. Hence, we use gradient information to further train the networks evolved by NEAT.

4.2 Training *type-1* networks

Type-1 networks are multi layer feed forward networks. They typically comprise of a set of sensory units (source nodes) that constitute the *input* layer, one *hidden* layer of computation nodes, and an *output* layer of computation nodes [Haykin, 1999]. The input signal propagates through the network in a forward direction, on a layer-by-layer basis. These neural networks are commonly referred to as *multilayer Perceptrons* (MLPs).

MLPs have been successfully applied to solve some difficult and diverse problems by training them in a supervised manner with a highly popular algorithm known as the *error back-propagation algorithm.*

We use a *conjugate gradient based back-propagation* method for training the MLPs, which is explained in detail in a later section

4.3 Training *type-2* networks

The following figure is a schematic representation of an Elman Network (*type-2* or ELM) that has been used for the purposes of this experiment.

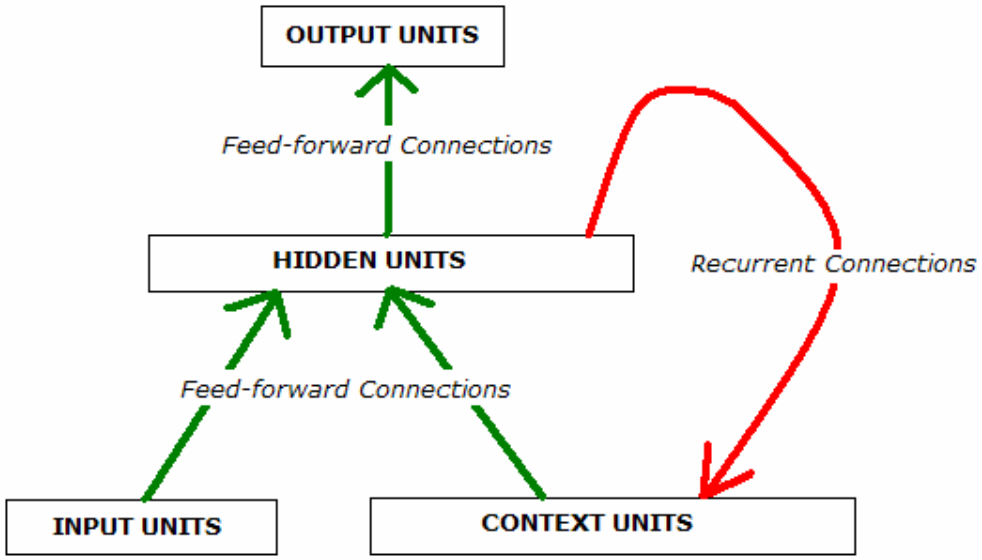

Figure 4.1: Elman Network

The Elman network [Elman, 1990] is a network having memory. This involves giving the processing system dynamic properties which are responsive to temporal sequences. Prior to the Elman networks appearing on the scene, a number of interesting ideas had appeared in literature [Jordan, 1986; Pineda, 1988; Williams and Zipser, 1989]. The Elman networks are a modification of Jordan's proposals.

The network shown in fig 4.1 is augmented at the input level by additional units, called *Context* units. These units are also "*hidden"* in the sense that they interact exclusively with other nodes internal to the network, and not the outside world.

Imagine that there is a sequential input to be processed and some clock which regulates presentation of input to the network. Processing would then consist of the following sequence of events. At time *t* the input nodes receive the first sequence of inputs. The *context* units are initially set to 0. Both the input units and the *context* units activate the hidden units; and then the hidden units feedforward to activate the output units. The hidden units also feed back to activate the *context* units, through the recurrent connections shown in figure 4.1. At the next time step, *t+1*, the above sequence is repeated. This time the *context* units contain values which are exactly the hidden unit activations at time *t*. These *context* units thus provide the network with memory.

After the presentation of the entire sequence, the network is trained using a *conjugate gradient* based *back-propagation of errors*, similar to the training of the MLP networks. Structurally the

ELM networks are very similar to the MLP networks, they just have an augmented input layer, and after every time step, the hidden layer activations are copied into the *context* units and hence are to become part of the input for the next time step. The recurrent connections are not adjusted during the *back-propagation* training, and are fixed at 1. The number of *context* units is equal to the number of the hidden units, and the upward connections between the *context* units and the hidden units were fully distributed, such that each *context* units activates all the hidden units.

4.3 Training *type-3* networks

The exact form of a gradient-following learning algorithm for completely recurrent networks was derived by [Williams and Zipser, 1989] and was demonstrated as the basis of practical algorithms for temporal supervised learning tasks. The algorithm didn't require a precisely defined training interval, and so could operate while the network ran. Their original algorithm called the *Real Time Recurrent Learning (RTRL)* algorithm, however requires a great deal of computation. It has a time complexity of $O(n^4)$, where n is the number of processing nodes in the Recurrent Neural Network (RNN). Another problem with RTRL is that the learning rate should be small to ensure smooth convergence. The consequence of a small learning rate is however that a large number of training epochs are required for the RNN to learn the temporal characteristics training patterns. To address this problem we incorporate *conjugate gradient* information computation into the RTRL algorithm, a slight variation of what is done in [Chang and Mak, 1999]. The resulting algorithm is denoted as the Conjugate Gradient Recurrent Learning (CGRL) algorithm.

Thus, we use *conjugate gradient* information to train each of the three different *types* of networks evolved by NEAT. In the next section we take a deeper look at the *Conjugate Gradient* method, and its application to MLP training. As ELMs are structurally similar to the MLPs, the same training approach is applicable to the ELMs as well. In a subsequent section we look at the CGRL algorithm in detail.

4.4 Conjugate Gradient Approach

First we shall describe the basic *back-propagation* of errors algorithm.

4.4.1 Back-propagation Algorithm

The error signal at the output of neuron *j* at iteration *n* (i.e., presentation of the *n*th training example) is defined by

$$e_j(n) = d_j(n) - y_j(n), \quad (4.1)$$

The instantaneous value of the error energy for neuron *j* is defined as ½ $e_j^2(n)$. Correspondingly, the instantaneous value ξ(n) of the total error energy is obtained by summing ½ $e_j^2(n)$ over all the neurons in the output layer; these are the only visible neurons for which the error signals can be calculated directly. We may thus write

$$\xi(n) = \tfrac{1}{2} \sum e_j^2(n), \ j \in C \quad (4.2)$$

where the set *C* includes all the neurons in the output layer of the network. Let *N* denote the total number of patters contained in the training set. The *average squared error energy* is obtained by summing ξ(n) over all n and then normalizing with respect to the set size N, as shown by

$$\xi_{av} = 1/N \ \sum \xi(n), \ n=1 \text{ to } N \quad (4.3)$$

For a given training set, ξ_{av} represents the *cost function* as a measure of learning performance. The objective of the learning process is to adjust the free parameters of the network to minimize ξ_{av}.

In the batch mode of back propagation learning, weight updating is performed *after* the presentation of *all* the training examples that constitute an epoch. For a particular epoch, the cost function is defined as the average squared error of Eq. (4.3). For a learning rate parameter η, the adjustment applied to synaptic weight w_{ji}, connecting neuron *i* to neuron *j,* is defined by the delta rule

$$\Delta w_{ij} = -\eta \ \partial \xi_{av} / \partial w_{ji} \quad (4.4)$$

$$= -\eta/N \ \sum_{n=1}^{N} e_j(n) \frac{\partial e_j(n)}{\partial w_{ji}} \quad (4.5)$$

The weight change $\Delta w_{ji}(n)$ is calculated as

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n) \ , \quad (4.6)$$

$$y_i(n) = \phi_i(v_i(n)) \quad (4.7)$$

$v_i(n)$ is the net input coming into node *i* and ϕ being the activation function of the node, now the local gradient $\delta_j(n)$ is defined by

$$\delta_j(n) = -\frac{\partial}{\partial v_j(n)}(\xi_{av}(n)) \quad (4.8)$$

According to Eq. (4.8), the local gradient $\delta_j(n)$ for output neuron *j* is given by the product of the corresponding error signal *ej(n)* and the derivative $\phi_j'(v_j(n))$, as in the next equation

$$\delta_j(n) = e_j(n)\phi_j'(v_j(n)) \quad (4.9)$$

When neuron *j* is located in a hidden layer of the network, there is no specified desired response for that neuron. Accordingly, the error signal for a hidden neuron would have to be determined recursively in terms of error signals of all the neurons to which that hidden neuron is directly connected. We may now define the local gradient $\delta_j(n)$ for hidden neuron *j* as

$$\delta j(n) = \phi_j'(v_j(n))\sum_k \delta_k(n) w_{kj}(n), \quad (4.10)$$

The factor $\phi_j'(v_j(n))$ involved in the computation of the local gradient in Eq. (4.9) depends solely on the activation function ϕ associated with hidden neuron *j*. The remaining factor involved in this computation, namely the summation over *k*, depends on two sets of terms. The first set of terms, the $\delta_k(n)$, requires knowledge of the error signals $e_k(n)$ for all neurons that lie in the layer just above the hidden neuron *j*, and all that are directly connected to hidden neuron *j*. The second set of terms, the $w_{kj}(n)$, consists of the synaptic weights associated with these connections.

4.4.2 Conjugate Gradient based Back Propagation

The conjugate gradient belongs to a class of second-order optimization methods collectively known as *conjugate-direction* methods. First we consider the minimization of the quadratic function

$$f(x) = \tfrac{1}{2}\, x^T A x - b^T x + c \quad (4.11)$$

where *x* is a *W*-by-1 parameter vector, A is a *W-by-W* symmetric, positive definite matrix, b is a *W*-by-1 vector and c is a scalar. Minimization of the quadratic function *f(x)* is achieved by assigning to *x* the unique value

$$x^* = A^{-1}b \quad (4.12)$$

Thus minimizing *f(x)* and solving the linear system of equations Ax*=b are equivalent problems.

Given the matrix A, we say that a set of non zero vectors *s(0), s(1),…,s(W-1)* is A-*conjugate* if the following condition is satisfied:

$$s^T(n)As(j) = 0 \quad \text{for all n and j such that } n \neq j \tag{4.13}$$

It can be easily shown that A-conjugate vectors are linearly independent. For a given set of A-conjugate vectors, the corresponding *conjugate direction method* for unconstrained optimization of the quadratic error function *f(x)* is defined by [Luenberger, 1973]

$$x(n+1) = x(n) + \eta(n)s(n), n = 0,1,...,W-1 \tag{4.14}$$

Where x(0) is an arbitrary starting vector, and $\eta(n)$ is a scalar defined by

$$f(x(n) + \eta(n)s(n)) = \min_{\eta} f(x(n) + \eta s(n)) \tag{4.15}$$

The procedure of choosing η so as to minimize the function $f(x(n) + \eta s(n))$ for some fixed n is referred to as a line search, which essentially is a one-dimensional minimization problem. The principal property of the conjugate direction method is that at successive iterations, it minimizes the quadratic function *f(x)* over an expanding linear vector space that eventually includes the global minimum of *f(x).*

For the conjugate direction method to work, we require the availability of a set of A-conjugate vectors s(0), s(1),…,s(W-1). In a special form of this method known as the *conjugate gradient method*, the successive direction vectors are generated as A conjugate versions of the quadratic function *f(x)* as the method progresses, hence the name of the method. Thus, except for n=0, the set of direction vectors {s(n)} is not specified beforehand, but rather it is determined in a sequential manner at successive steps of the method.

We define the *residual* as the steepest descent direction

$$r(n) = b - Ax(n) \tag{4.16}$$

Then to proceed, we use a linear combination of r(n) and s(n-1) as shown by

$$s(n) = r(n) + \beta(n)s(n-1), \qquad n = 1,2,\ldots,W-1 \tag{4.17}$$

where $\beta(n)$ is a scaling factor to be determined. Multiplying this equation by A, taking the inner product of the resulting expression with s(n-1), invoking the A-conjugate property of the direction vectors, and then solving the resulting expression for $\beta(n)$, we get

$$\beta(n) = -\frac{s^T(n-1)Ar(n)}{s^T(n-1)As(n-1)} \tag{4.18}$$

Generation of the direction vectors in accordance with the recursive equation (4.17) depends on the coefficient $\beta(n)$. The formula in equation (4.18) for evaluating $\beta(n)$, as it presently stands, requires knowledge of matrix A. For computational reasons it would be desirable to evaluate $\beta(n)$ without explicit knowledge of A. This evaluation can be achieved by using one of two formulas [Fletcher, 1987]:

1. Polak-Ribiere formula

$$\beta(n) = \frac{r^T(n)(r(n) - r(n-1))}{r^T(n-1)r(n-1)} \tag{4.19}$$

2. Fletcher Reeves formula

$$\beta(n) = \frac{r^T(n)r(n)}{r^T(n-1)r(n-1)} \tag{4.20}$$

To use the conjugate-gradient method to attack the unconstrained minimization of the cost function $\xi_{av}(w)$ pertaining to the unsupervised training of multilayer Perceptrons, two things are to be done

1) Approximate the cost function by a quadratic function
2) Formulate the computation of coefficients $\beta(n)$ and $\eta(n)$ in the conjugate gradient algorithm so as to only require gradient information.

Finally, a summary of the conjugate gradient algorithm for the supervised training of a multilayer perceptron is given below:

1)For w(0), initial weight vector, use back-propagation to compute the gradient vector g(0)

2) Set s(0) = r(0) = -g(0)

3) At time step *n,* use a line search (like *Brent's method* that minimizes the cost function $\xi_{av}(\eta)$ significantly, when the cost function is expressed as a function of η for fixed values of w and s.

4)Test to determine if the Euclidean norm of the residual r(n) has fallen below a specified value, that is, a small fraction of the initial value $\|r(0)\|$

5) Update the weight vector:

$$w(n+1) = w(n) + \eta(n)s(n)$$

6) For w(n+1), use back propagation to compute the updated gradient vector g(n+1)

7) Set r(n+1) = -g(n+1)

8) Use the Polak-Ribiere method to calculate $\beta(n+1)$

$$\beta(n+1) = \max\{\frac{r^T(n+1)(r(n+1)-r(n))}{r^T(n)r(n)}, 0\}$$

9) Update the direction vector

$$s(n+1) = r(n+1) + \beta(n+1)s(n)$$

10) Set n=n+1, and go back to step 3

Stopping Criterion:

Terminate the algorithm when the following condition is satisfied

$$\|r(n)\| \leq \varepsilon\|r(0)\|$$

where ε is a prescribed small number

4.5 Conjugate Gradient Method for the RTRL Networks

An RTRL network, as shown in the following figure, consists of two layers, and input layer with *m* nodes (referred to as the input layer), and an RTRL layer with *n* nodes (referred to as the RTRL layer).

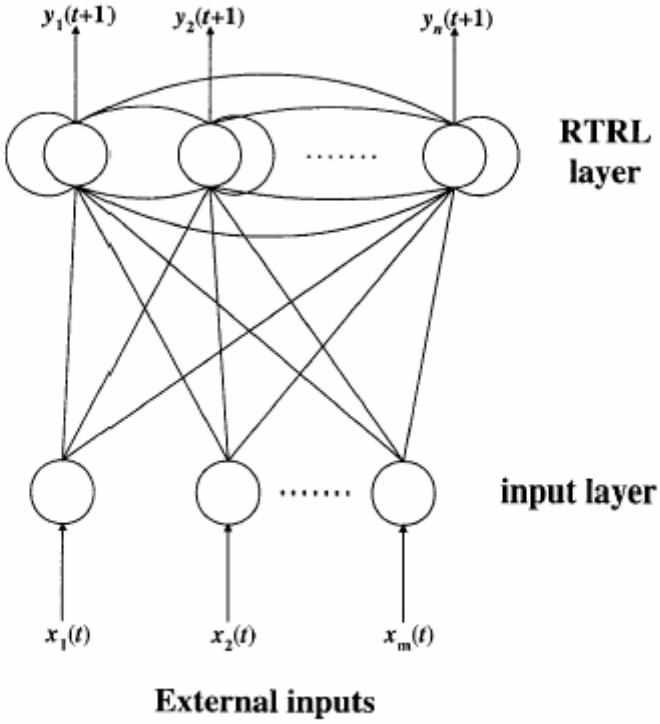

Figure 4.2: An RTRL network

Let *y(t)* denote the *n*-tuple of outputs of the RTRL nodes at time t and *x(t)* the m-tuple of external inputs to the network at time t. We concatenate *x(t)* and *y(t)* to form the *(m+n)*-tuple *z(t)*, with *U* denoting the set of indices of the RTRL nodes and *I* the set of indices for the external inputs so that

$$z_k(t) = \{x_k(t), k \in I; y_k(t), k \in U\} \tag{4.21}$$

By adopting the indexing convention just described, all weights can be incorporated into an *n x (m + n)* matrix *W*. Now, denote,

$$s_k(t) = \sum_{l \in U \cup I} w_{kl} z_l(t), k \in U \tag{4.22}$$

as the net input to the *k*th unit at time t. Thus the output at the next time step becomes

$$y_k(t+1) = f_k(s_k(t)) \tag{4.23}$$

where $f_k()$ is an activation function. The external inputs at time *t* do not influence the outputs of any unit until time *t+1.*

Denote $\theta(t)$ as the set of indices $k \in U$ for which a specified target $d_k(t)$ for the output of the *kth* unit exists. Then, we define $e_k(t)$ to be the *n*-tuple of the error at time *t.*

$$e_k(t) = \{d_k(t) - y_k(t), k \in \theta(t); 0, otherwise\} \tag{4.24}$$

Now let,

$$E(t) = \frac{1}{2}\sum_{k \in U}[e_k(t)]^2 \tag{4.25}$$

which denotes the overall error at time *t.* Assume that the network starts to operate at time t_0 and continues to operate upto some time t_1. The total squared error over this trajectory is given by

$$E_{total}(t_o, t_1) = \sum_{t=t0+1}^{t1} E(t) \tag{4.26}$$

The objective is to minimize the total error by computing its first derivative, given by,

$$\nabla_w E_{total}(t_0, t_1) = \sum_{t=t0+1}^{t1} \frac{\partial E(t)}{\partial w_{ij}} \tag{4.27}$$

Therefore the total weight change is given by

$$\Delta w_{ij}(t_0, t_1) = \sum_{t=t0+1}^{t1} \alpha \sum_{k \in U} e_k(t)\, p_{ijk}(t) \tag{4.28}$$

where α is the learning rate, and

$$p_{ijk}(t) = \frac{\partial y_k(t)}{\partial w_{ij}} \tag{4.29}$$

is a parameter reflecting the importance of the link between node *j* and node *i* on the output value of node *k*, and

$$\Delta w_{ij}(t) = \alpha \sum_{k \in U} e_k(t)\, p_{ijk}(t) \tag{4.30}$$

is the instantaneous weight change at time t.

$$p_{ijk}(t+1) = f_k'(s_k(t))\{\sum_{l \in U} w_{kl} p_{ijl}(t) + \delta_{ik} z_j(t)\}, \tag{4.31}$$

where $i \in U, j \in U \cup I, k \in U, f'()$ is the derivative of $f()$, δ_{ik} is the Kronecker delta and $p_{ijk}(t_0) = 0$.

The RTRL training algorithm implemented for the purposes of this project, followed the *Batch-mode,* i.e. all the training errors were accumulated, and changes to the weights were made only after the entire training set had been presented to the network. A complete presentation of all training data is called a training epoch. At the beginning of each epoch p_{ijk} and Δw_{ij} are set to zero for all *i,j and k*. They are calculated at each time step (Eq. 4.30 and Eq. 4.31) and accordingly incremented.

The approach explained above is the original RTRL algorithm as proposed by [Williams and Zipser, 1989]. It is a "greedy" approach, and takes many false steps, as it only uses the first derivative and therefore doesn't always take the most direct path. Hence, we used a conjugate gradient based modification of the above algorithm for better training and convergence.

The objective function to be minimized is now defined as

$$J(w) = \frac{1}{N}\sum_{n} E_n \tag{4.32}$$

where *N* is the number of patterns in the training set. E_n is the output error for the *nth* pattern, and *w* is the weight vector. E_n defined as

$$E_n = \frac{1}{N}\sum (d_{kn} - o_{kn}(w))^2 \text{ , summed over all } k \tag{4.33}$$

where o_{kn} and d_{kn} are the actual and desired outputs of the *kth* output unit in response to the *nth* input pattern respectively. To compute the gradient of the objective function, we differentiate Eq.4.32 with respect to *w*, to obtain the gradient $g(w)$ as

$$g(w) = \frac{1}{N}\sum_{n} \nabla E_n(w) = \frac{1}{N}\sum_{n} (-\sum_{k} e_k(n)\, p_{ijk}(n)) \tag{4.34}$$

Once the gradient and objective functions for the RTRL networks has been identified, the conjugate gradient algorithm explained in the previous section can be used to train the RTRL networks.

In addition to the three dimensional matrix p_{ijk}, CGRL needs to store the error and gradient information of the points used in the line search of the conjugate gradient method. As a result of this, its space complexity is higher than that of the RTRL algorithm. Despite these limitations, the convergence performance of this method is still better than the RTRL algorithm [Chang and Mak, 1999].

Chapter 5

Combining Neural Networks

Ensembles of artificial neural networks show improved generalization capabilities that outperform those of single networks. However, for aggression to be effective the individual networks must be as diverse and accurate as possible. In this chapter, we explain how the networks evolved by NEAT, and trained by Conjugate Gradient based back propagation were aggregated in a way that required the least computation, and resulted in performances that were better than those of any of the individual networks in the aggregate.

5.1 Aggregating Neural Networks

For most regression and classification problems combining the outputs of several predictors improves on the performance of a single generic one [Sharkey, 1999]. Formal support to this property is provided by the bias-variance dilemma [Geman *et al.,* 1992]. According to these ideas, good ensemble methods must be both *accurate* and *diverse*, which poses the problem of generating a set of predictors with reasonably good individual performances and independently distributed predictions for the test points.

Diverse individual predictors can be obtained in various ways. Ways to do that include (1) changing the internal structure of the neural networks, (2) learn from different adequately chosen subsets of the data set.

Two extensively used methods for ensemble construction are (a) *bagging* [Breiman, 1996] and (b) *boosting* [Freund and Schapire, 1995]. Bagging (short for 'bootstrap aggregation'), and variants thereof, train independent predictors on bootstrap re-samples Ln (n = 1,M) of the available data D, usually employing the unused examples $Vn = D - Ln$ for validation purposes. These predictors are then aggregated according to different rules (for instance, simple or weighted average). Boosting and its variants are stagewise procedures that, starting from a predictor trained on D, sequentially train new aggregate members on bootstrap re-samples drawn with modified probabilities. According to the general approach, each example in D is given a different chance to appear in a new training set by prioritizing patterns poorly learnt on previous stages. In the end, the predictions of the different

members so generated are weighted with a decreasing function of the error each predictor makes on its training data.

In a time series forecasting task, resampling the data as required by both bagging and boosting is not possible. The network has to be trained on the entire temporal sequence for it to be able to generalize it. Additionally, these methods solely focus on the data that a network has been trained on, and not at its architecture. In the next section we shall see how networks evolved by NEAT can be effectively used as members of an ensemble for obtaining superior performances.

5.2 Creating an Ensemble using Networks evolved by NEAT

NEAT evolves the topology and weights of the neural networks simultaneously. Networks evolved by NEAT for supervised learning tasks have been topologically optimized, and have achieved a certain level of precision through the various weight mutations. After being further optimized using the gradient descent approach, the networks have been fine-tuned for performance. This accounts for the *accuracy* that ensemble members need for the ensemble to be better than any of the individual networks.

Diversity of the networks evolved by NEAT was ensured through two different processes, (1) repeated runs of NEAT produced topologically diverse networks, as NEAT, by its very philosophy carries out a stochastic search in both the weight and topology space of the networks to find the best solutions, (2) the input data was put through different normalizations and smoothing functions, or augmented using other relevant parameters to cause NEAT to evolve networks that would learn to generalize using different input parameters.

The networks evolved by NEAT, after being trained using the conjugate gradient training algorithm till they have achieved certain accuracy, were thus extremely suitable as members of an ensemble, and as we shall see in a later chapter, the prediction capability of an ensemble thus formed was much better than those of any of the individual networks. Additionally, this also is being presented as a novel method of building meaningful ensembles to improve precision in supervised learning tasks.

5.3 The Ensemble

Earlier work on building ensembles for regression tasks have proposed many interesting ideas [Zhou *et al.* 2002; Granitto *et al.*, 2005]. However, all the methods start out with a number of neural

networks, and then apply some methods to *select* the best networks to be included in the ensemble. We shall soon see how NEAT can overcome this problem, making it a very straightforward process to build an ensemble. It has also been observed that instead of simply averaging the ensemble members, taking a weighted sum of their averages actually improves performance [Granitto *et al.*, 2005]. This has also been incorporated into the ensemble that we build using the networks evolved by NEAT.

5.3.1 Selecting the Ensemble Members

The networks evolved by NEAT are already topologically optimal. Further training them with the conjugate gradient algorithm improves their performance to a great extent. To improve the *diversity* of the networks, the training data used for the different networks is made slightly different. Different runs of NEAT begin with data that has been normalized/smoothed in different ways. It is known that networks trained on different training sets implicitly develop different generalizing capabilities. This augurs well for our ensemble, as greater diversity will lead to better ensemble predictions. The initial diversity in the starting population of NEAT also plays a role in this, instead of 8 input nodes in one run of NEAT, we might start with 9 in the next one. This makes it easier for NEAT to bring about topological diversity, though it must be kept in mind that it also causes NEAT to search in higher-dimensional spaces, and may at times do more harm than good.

Thus *NEAT+gradient descent training* gives us networks that are ready to be part of the ensemble. It isn't necessary to apply any other algorithm to create the ensemble members. This is in contrast to most other ensemble creation methods. Slight variations in the training data combined with moderate diversity in the starting population, gives NEAT followed by a conjugate gradient based training process, the ability to directly produce networks that can either be used independently, or else combined to form an ensemble for still better performance.

5.3.2 Weighing the Ensemble Members

It is known that taking a weighted sum of the ensemble members outputs' results in better performance, as versus simply averaging their outputs. In earlier work, following general ideas from *boosting*, the output of the ensemble was given as

$$\Phi_m(x) = \sum_{n=1}^{m} w_n f_n(x) \qquad (5.1)$$

where w_n is a decreasing function of e_n, the MSE of the *n*th predictor in the ensemble. This is the way in which boosting reduces the

importance of over fitted or badly performing predictors included in the ensemble.

While making our ensemble, we do not just take the outputs of the individual ensemble members, rather we take both the hidden layer outputs *and* the activations of the output nodes of each of the ensemble members into account. So we form a weighted sum of the hidden layer activations (RTRL layer activations in the case of the *type-3* networks) and the output layer activations. This weighted sum is considered the final output of our ensemble. So the final error to be minimized was

$$MSE_{train} = \frac{1}{N}\sum_{n=1}^{N}(d(n) - w_{out}x(n))^2 \qquad (5.2)$$

where $x(n)$ is a vector containing the activations of the hidden and output layers of all the ensemble members. The weight vector w_{out} that minimizes this MSE is made from the regression weights of the linear regression of the $d(n)$ values on the $x(n)$ values, as in [Jaeger and Haas, 2004]. Here the $d(n)$ values are the target values in the supervised learning task.

Chapter 6

Experiments and Results

Our first goal was to see if NEAT could evolve minimal networks for a complex supervised learning task. [Whiteson *et al.*, 2004] had used a version of FS-NEAT to solve a line-orientation task. We tried to evaluate NEAT's capabilities on a chaotic time series prediction task, the Mackey-Glass time series [Mackey and Glass, 1997].

6.1 Mackey Glass Time Series

The Mackey-Glass (MG) time series represents a model for white blood cell production in leukemia patients and has nonlinear oscillations which is widely used for testing the performance of neural network models. The series is a chaotic time series which makes it a universally acceptable representation of nonlinear oscillations of many physiological processes. The MG delay-difference equation is given by

$$y(t+1) = (1-b)y(t) + a\frac{y(t-\tau)}{1+y^{10}(t-\tau)} \qquad (6.1)$$

where $a = 0.2, b = 0.1, \tau = 17$. The time delay τ is a source of complications in the nature of the time series.

NEAT started with an initial population consisting of the three *types* of networks, each *type* having a hundred individuals in a population composed of 300 neural networks. The *type-3* networks all had 5 input nodes and 1 output node. All the input nodes were connected to the output node. The *type-1* and *2* networks, started off with 5 input nodes, 2 hidden nodes and 1 output node. Connectivity was complete, in the sense the input layer was completely interconnected with the hidden layer, and the hidden layer in turn with the output neuron. The *type-2* and *type-1* networks additionally had biases too. All the connection weights were initialized to lie between 0.5 to -0.5.

The purpose of the modelling was to do one step prediction, *i.e.,* in the case of the starting population it was to take in $y(t), y(t-1), y(t-2), y(t-3), y(t-4)$ as inputs, and use them to predict $y(t+1)$. The training set composed of points 500-1000 of the series, and the networks performance was to be evaluated on the next set of 100 points, i.e., 1001-1100.

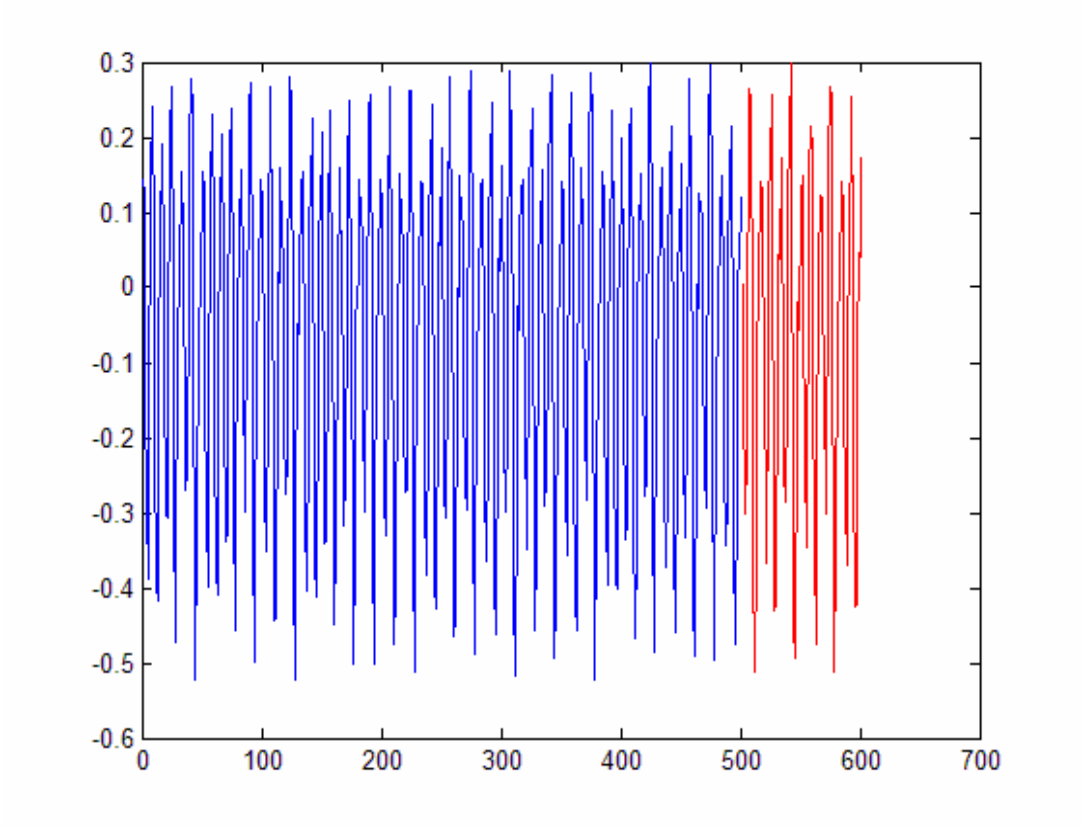

Figure 6.1: MG Data set

In the above figure the MG data has been plotted (after it has been normalized between 0.3-0.6. The blue part is the training set, and the red portion is the testing set.

Of the 5 instances of NEAT that were started on the MG data set, 3 runs produced RTRL networks as the best networks, the other two produced a MLP network and an ELM network respectively.

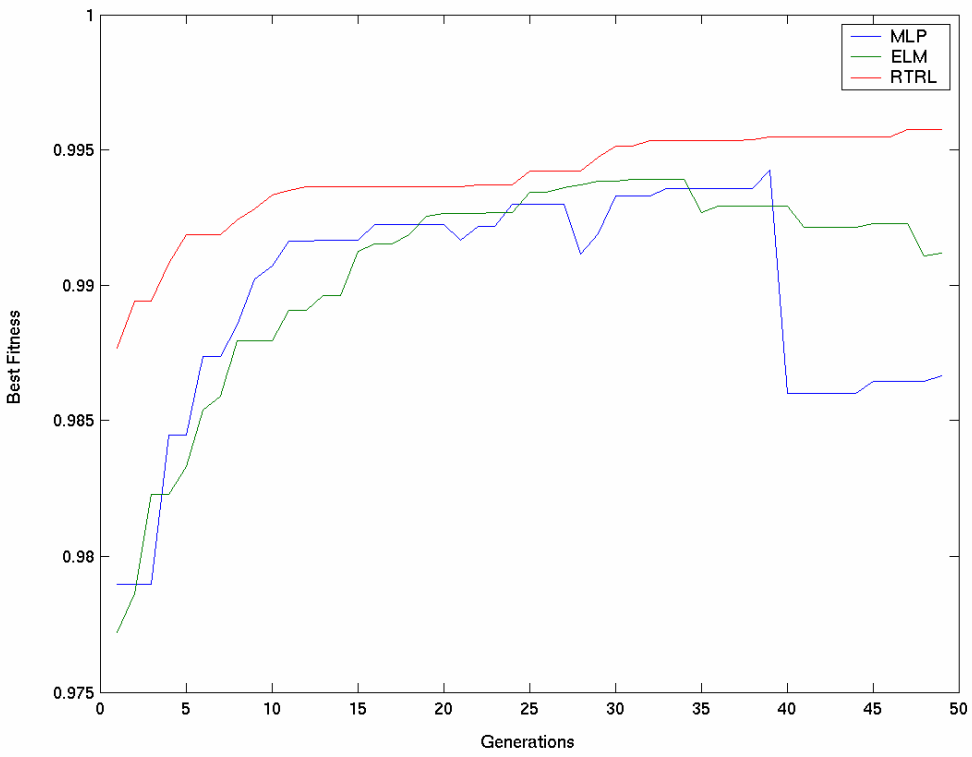

Figure 6.2: The Generations vs. Best-Fitness Graph

In the above figure we see how the best individuals of each *type* perform as evolution proceeds. It can be seen that the RTRL networks (red) dominate the evolution throughout the process. The ELM (green) and MLP (blue) networks show continuous improvement, but are not able to beat the RTRL networks.

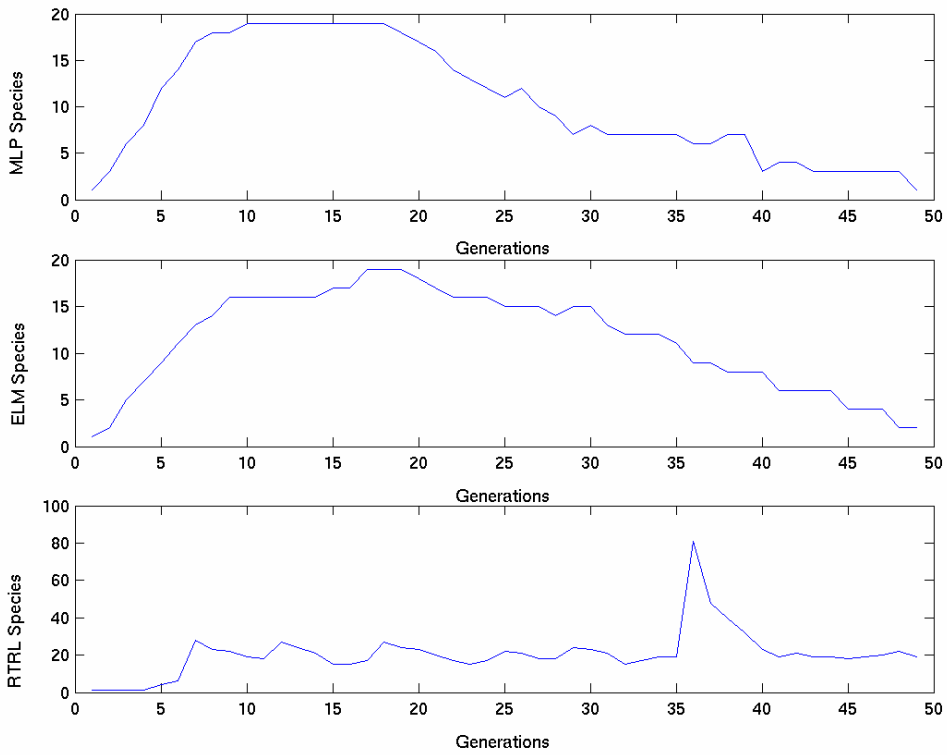

Figure 6.3: The Species Graph

The above figure shows the number of species of the three *types* of networks throughout evolution. A high value of the speciation threshold set initially prevents the RTRL population from growing out of bounds, however we still see a sharp spike at around the 36th generation, where the number of RTRL species reached around 80, but it soon comes down as the speciation threshold is adaptive in nature. In case of the MLP/ELM networks, evolution gives them an initial space to grow, however to prevent evolution from being slowed down, it later on clamps down on the MLP/ELM networks, and we see their species numbers coming down.

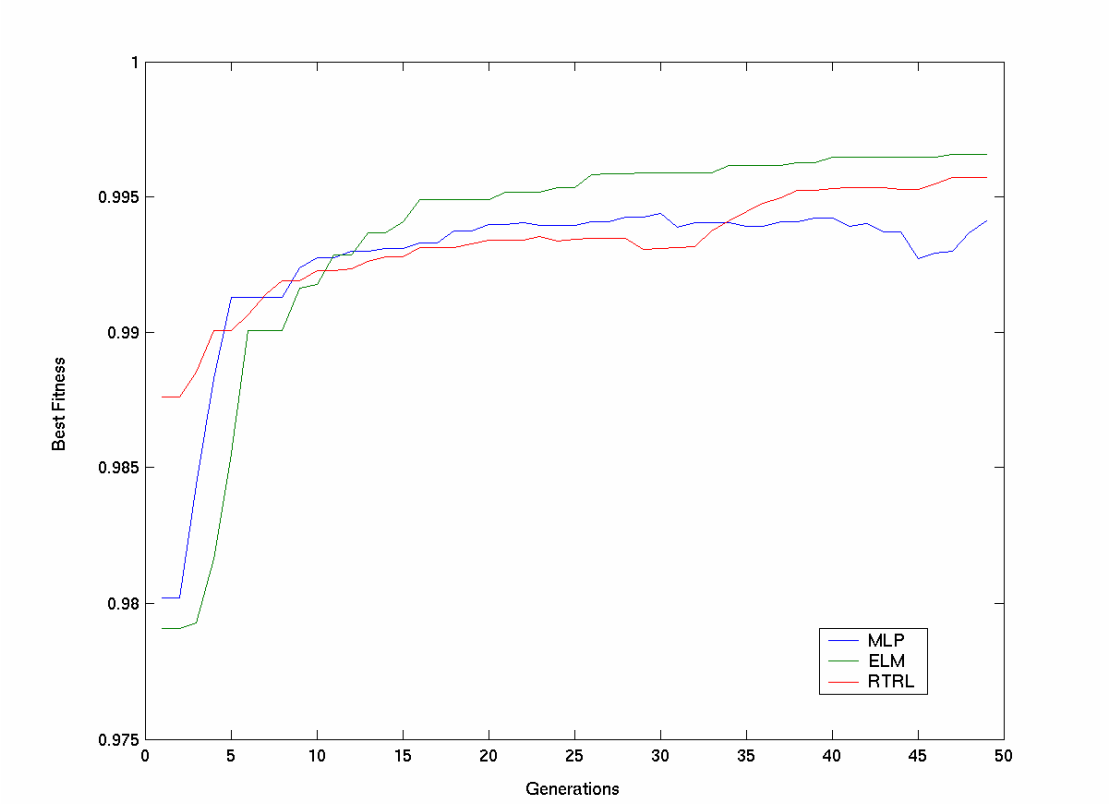

Figure 6.3: ELM network dominating

In figure 6.4 we see the ELM networks dominating evolution to produce the network of the best fitness.

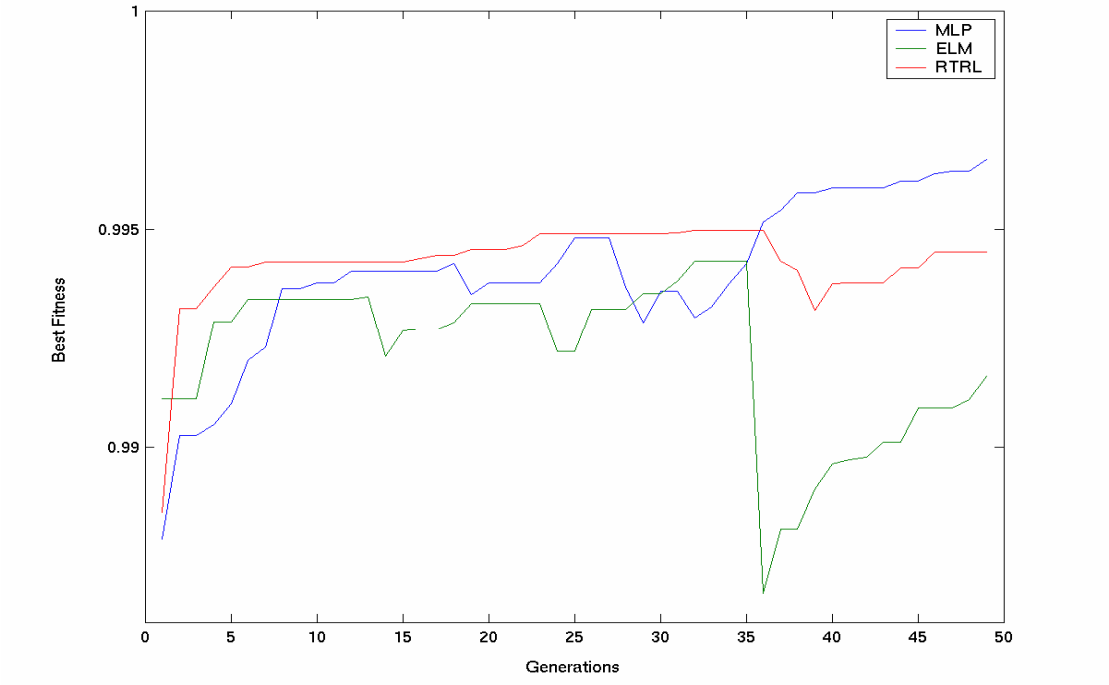

Figure 6.4: MLP network dominating

In figure 6.5 we see a MLP network being the best network in evolution. This clearly demonstrates NEAT's ability to create *diversity* in the population. All the networks had a MSE of around 3e-3, and so had been trained to about the same precision. But the fact that they were all of different architectures clearly shows NEAT's ability to create diversified networks. The best network evolved by NEAT for the MG data set was a *type-3* network having 5 input nodes and 2 RTRL nodes.

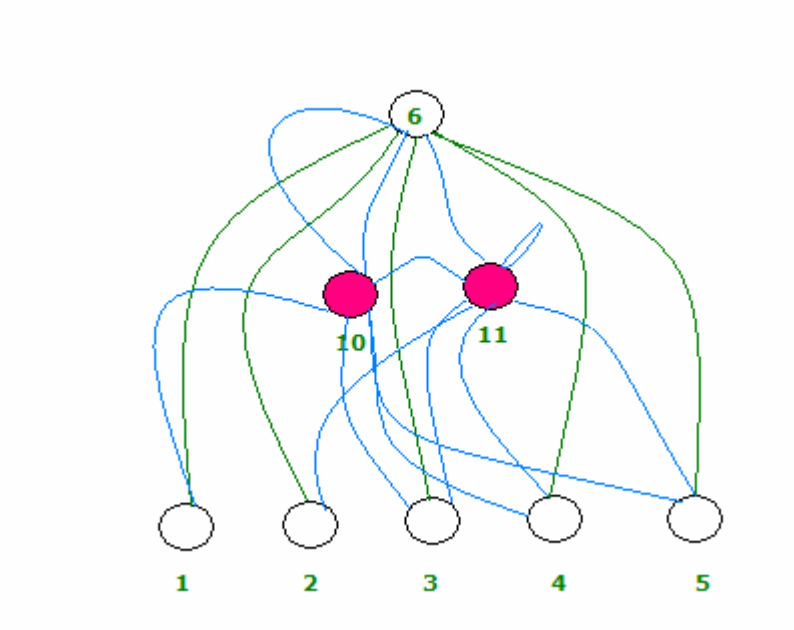

Figure 6.5: The Network Evolved for the MG Data set

The above figure shows the network evolved for the MG data set by NEAT. It had a training MSE of 3.3e-3. The population had started with networks having only 5 input nodes and 1 output node. We see that NEAT has added 2 hidden nodes (shown in red) and 14 new connections (shown in blue). The two way connections have not been shown in the figure.

The above network was trained using the CGRL algorithm explained in section 4.5 for about 3000 epochs. The training MSE for the final network was calculated to be 6.4025e-4 and the test MSE was calculated to be 7.4111e-4, where MSE is calculated as

$$MSE = \frac{1}{2N}\sum_{n=1}^{N}(t(n)-o(n))^2 \qquad (6.2)$$

The performance of the trained network on the test set is shown on the following figure.

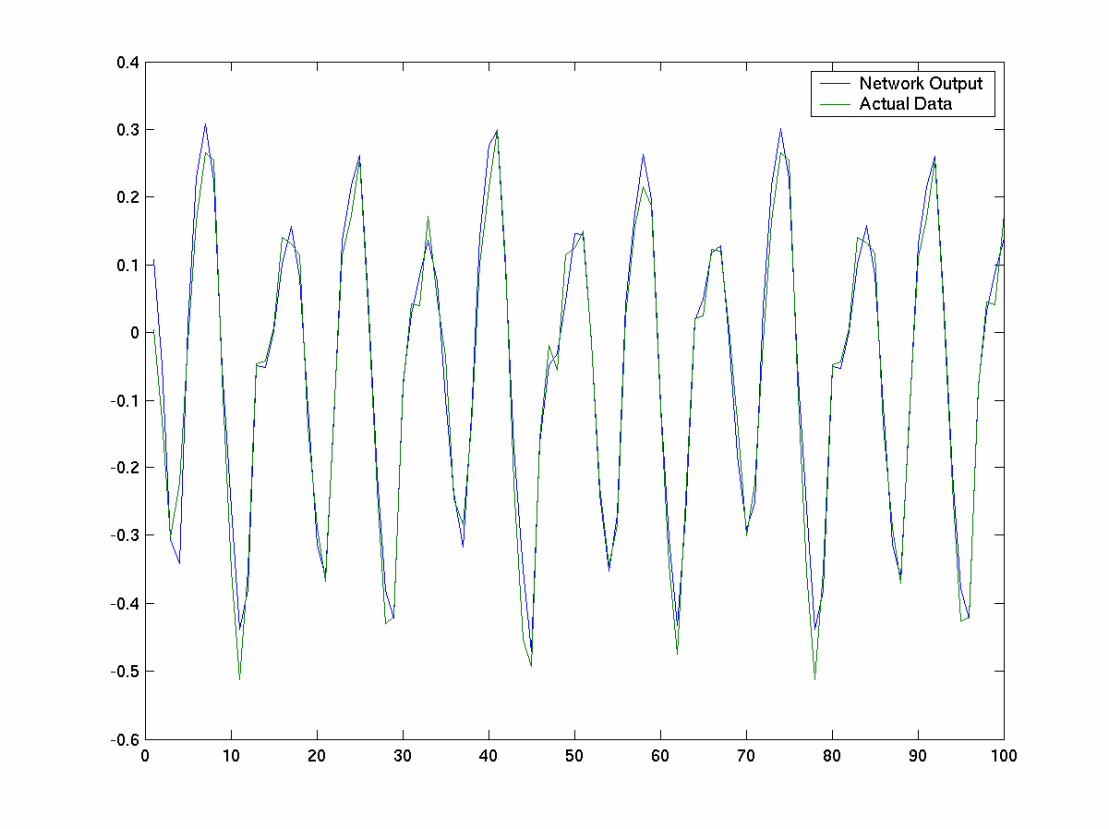

Figure 6.6: Trained networks performance on the test set

Thus it is seen that the one step predictive ability of the network is reasonably good, and is better than some other results reported in literature, like [Yadav *et al.* 2006]. We did not do the ensembling for this experiment as the goal of this experiment was to just evaluate NEAT's ability to evolve networks for complex supervised learning tasks. The results confirmed that NEAT could do it, as it had been reasonably successful in evolving networks for a chaotic time series prediction task. This experiment also establishes NEAT as an evolutionary paradigm that creates networks with minimal structure that can solve problems. The best results for this data set have been obtained by [Jaeger and Haas, 2004] using a special network, called an Echo State Network, that uses a *reservoir* of 1000 recurrent neurons for a 2000 point training set.

6.2 Predicting Currency Exchange Rates

Our second experiment concerned evolving neural networks to predict currency exchange rates. The daily Yen vs. Dollar exchange rate data pertaining to the period 1990-1995 was collected from the Federal Bank of St Louis. The first 1000 points of the set were to be used for the training, and the remaining 100 points were to be part of the test set.

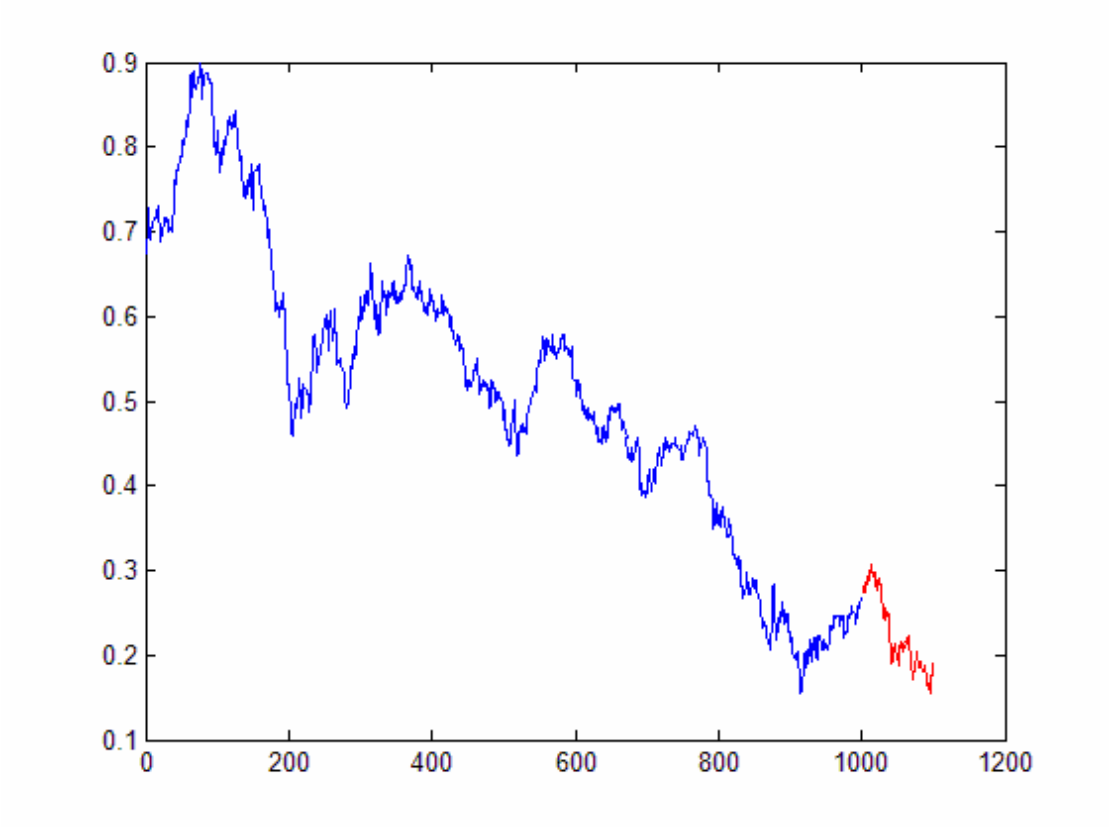

Figure 6.7: The Currency data set, blue training and red test

In this experiment, to implement ensembling, we instantiated 4 different runs of NEAT, each with a certain diversity in the population, and each setup being made to use a slightly different data set. The data sets used for the training purposes, as has been mentioned previously, were developed by smoothing/augmenting the original data set. The original data set, that of the daily Yen vs. Dollar exchange rate, was normalized to lie between 0.1-0.9. Run 1 used three EPMA inputs apart from the normalized exchange rate, where EPMA inputs are defined as

$$\mu_n = \left[\frac{2}{p(p+1)}\right]\sum_{i=1}^{p} x_{(n-i+1)}(3i-p-1) \qquad (6.3)$$

where p is called the *span* of the EPMA calculation. In Run1 the spans used were 5, 10 and 15 days. Run 2 also augmented its input by three EPMA values, having spans of 5, 15 and 30 days. Run 3 used EPMA values of span 30 only as its inputs, and in Run 4, the input was the normalized exchange rate along with two EPMA inputs, of spans 5 and 10 respectively.

The *WINNER* networks for the Runs 1 and 2 were selected by the end of the 10th and 11th generation respectively. In the case of Runs 3 and 4, the best networks were selected at the end of the

20th generation. The various parameters of the NEAT process for the currency exchange rate prediction task are listed in Appendix A.

In the following paragraphs we shall see the fitness graphs of some of these runs, as well as the after gradient training performances of the winning networks in each of the 4 runs. Table 6.1 lists the configuration of the starting population and the *Winner* network's *type* in each of the 4 runs of the NEAT algorithm. Table 6.2 details the structures of the *Winner* networks.

Table 6.1: Starting configuration of the NEAT population and the *type* of network finally evolved

Run	**No of Input nodes in Start. Pop.**	**No. of Hidden nodes (ELM/MLP) In Start Pop.**	***type* of Best Network evolved by NEAT**
1	10	2	RTRL
2	8	2	RTRL
3	5	2	RTRL
4	8	2	ELM

Table 6.2: Configuration of the *WINNER* neural networks at the end of the NEAT process

Run	**No of Inputs**	**No of Hidden nodes**	**MSE at the end of NEAT based evolution**
1	10	4	5e-3
2	11	4	4.9e-3
3	7	2	8e-4
4	10	8	5e-4

Table 6.3: MSE_{train} and MSE_{test} at the end of the Conjugate Gradient based training process for the *WINNER* networks

Run	**MSE at the end of NEAT**	**MSE at end of CG training**	**MSE on Test set**
1	5e-3	1.5e-3	4.8e-3
2	4.9e-3	5.9e-4	1.3e-3
3	8e-4	1.97e-4	1.6e-4
4	5e-4	5.19e-5	4.08e-5

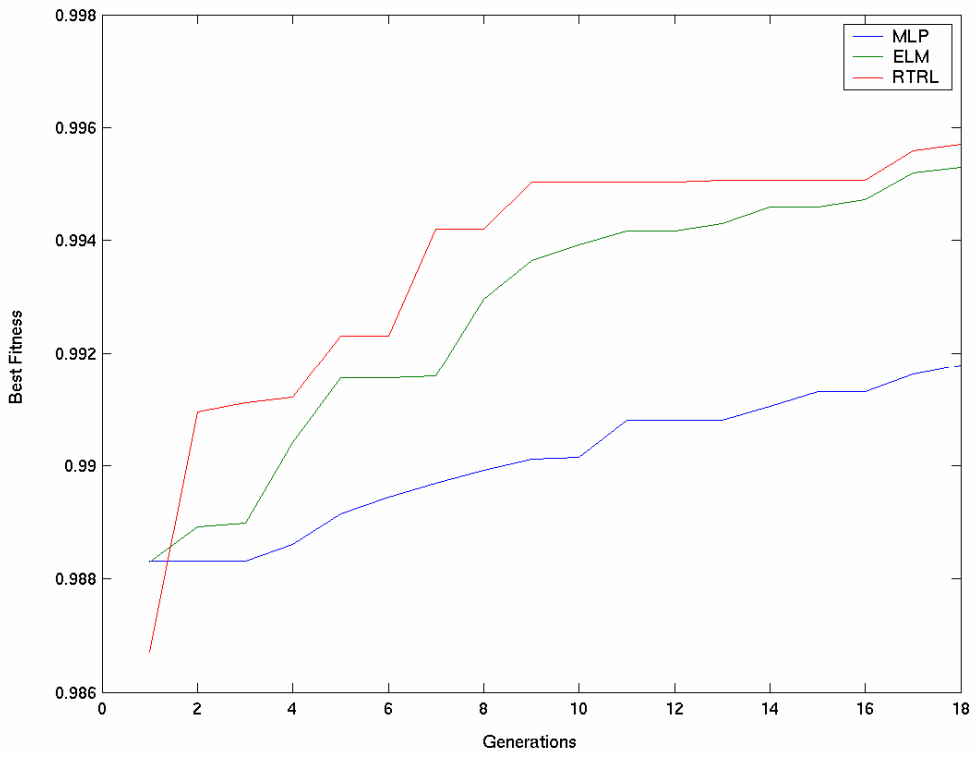

Figure 6.8: Fitness Vs Generation graph of Run 1

The Winner network in Run 1 was selected at the end of the 10^{th} generation itself. It is seen that the RTRL network that dominates was the poorest network to begin with. By the second generation however, NEAT has given it the structure necessary to outperform the ELM/MLP networks in the population.

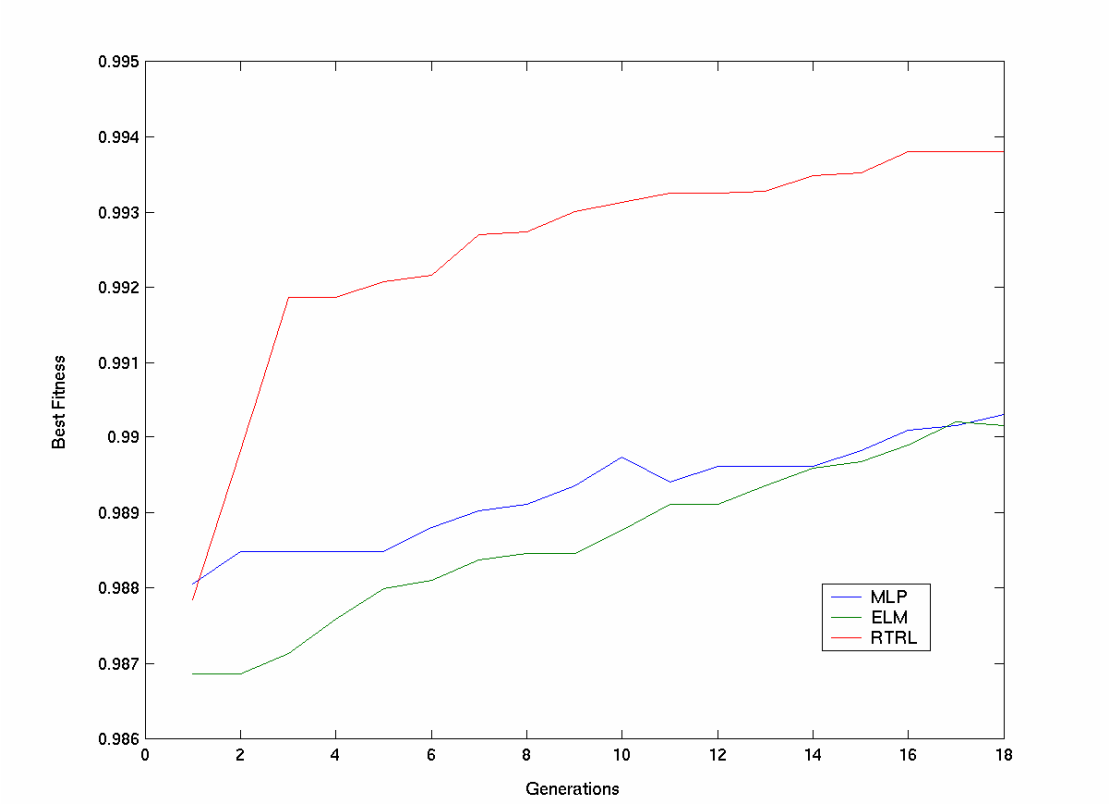

Figure 6.9: Fitness vs. Generations graph of Run 3

The above figure shows the Fitness vs. Generations graph of Run 3. Here again we see the RTRL network dominating the evolutionary process.

6.2.1 The Random Walk model

As mentioned in Chapter 1, a number of studies have said that currency exchange rates follow a random walk. The Random Walk model says that currency exchange rates are inherently unpredictable, i.e. they follow a random walk. If the closing rate today is $y(t)$, then tomorrow's closing rate $y(t+1)$ is simply given by

$$y(t+1) = y(t) + \varepsilon \tag{6.4}$$

where ε is nothing but white noise. Accepting this model, the best prediction for the next day's closing rate is simply today's closing rate. Most Statistical models have not been able to improve upon the predictions of the Random Walk Model. A test for the efficiency of any exchange rate forecasting process thus is whether it can improve upon the predictions of the Random Walk Model.

6.3 Ensemble Performance

The RW model gives an MSE of 3.7113e-5 on the test set. Table 6.3 shows that none of our individual networks is able to beat the RW model on the test set. We now look at the results of the ensembling process which was explained in section 5.3.2. A set of weights w_{out} satisfying Eq. 5.2 was calculated using Linear Regression. Each of the 4 networks from the NEAT runs, after being put through gradient descent training, was made part of the ensemble. The Ensemble thus created produced a MSE_{train} of 5.22e-5 and a MSE_{test} of 3.582e-5, which clearly outperforms the Random Walk Model.

It was also observed that with the addition of the each network evolved by NEAT and trained by gradient descent to the ensemble, its performance improved. This was verified by starting with just 2 networks, those produced by Runs 4 and 3, and then calculating performance of the ensemble on the test set after the addition of each new network. First the network produced by Run 1 is added to the ensemble, followed by that produced by Run 2. The results are tabulated in Table 6.4.

Table 6.4: Ensemble performance with the sequential addition of additional networks to it

Number of networks in ensemble	MSE on the test set
2	3.8507e-5
3	3.6766e-5
4	3.5820e-5

Thus we see an ensemble of networks evolved by NEAT, and trained using a powerful conjugate gradient based back propagation method shows better performance than any of the individual networks, and is also able to beat the Random Walk model on a complex financial time series forecasting task

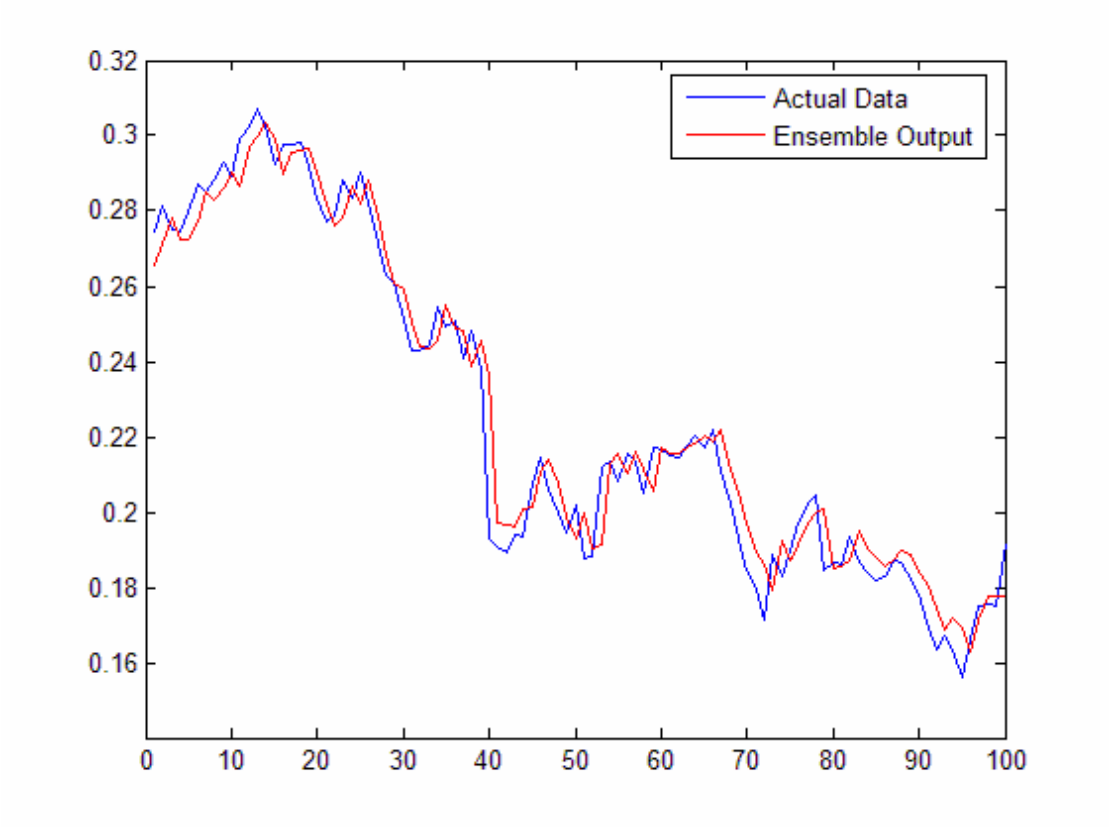

Figure 6.10: Ensemble performance on the Test set

Chapter 7
Conclusions

NEAT presents several advances in the evolution of neural networks. It minimizes structure throughout evolution, it incrementally elaborates structure in a stochastic manner from a minimal starting point. Because of speciation, useful elaborations survive even of they are initially detrimental. Thus, NEAT strengthens the analogy between GAs and natural evolution by not only performing the optimizing function of evolution, but also a *complexifying* solution, allowing solutions to become incrementally more complex at the same time as they become more optimal. It is this complexifying function that makes NEAT unique among GAs.

NEAT has been applied successfully to many control learning tasks like the benchmark Pole Balancing tasks, on which its performance has been better than other NeuroEvolutionary/Reinforcement Learning methods.

The success of NEAT coupled with gradient descent and linear regression, now establishes NEAT as the evolutionary method of choice for solving complex supervised learning tasks. Evolino [Schmidhuber *et al.*, 2005] is another evolutionary system that makes use of gradient information to solve supervised learning tasks, and has arrived on the scene rather recently. However, for evolution it employs *Enforced Sub Populations* (ESPs). NEAT has been demonstrated to be much faster than ESP [Stanley and Miikkulainen, 2002]. On a complex pole balancing task, NEAT has been shown to be 5 times faster than ESP. This certainly makes a NEAT a method of choice for neuro-evolutionary tasks.

NEAT's ability to create diversity through topological mutations and Conjugate Gradient (CG) based Back-Propagation's (BP's) ability to bring about accuracy and precision make NEAT combined with CG-BP a novel and powerful way to create ensembles, whose performance will be better than each of the independent generalizers that comprise it. Ensemble building using NEAT is simple and effective. For temporal sequences, wherein we cannot sample our data and hence methods like *bagging* and *boosting* are not applicable, a NEAT based approach followed by gradient descent training certainly is a reliable method to build powerful ensembles of neural networks.

References

Alexander D., Thomas L. R., 1987, "Monetary/Asset Models of Exchange Rate Determination: How Well Have They Performed in the 1980s?", **International Journal of Forecasting**, Vol 3, pp 53-64

Angeline P. J., Saunders G. M., and Pollack J. B. (1993), "An evolutionary algorithm that constructs recurrent neural networks", **IEEE Transactions on Neural Networks**, 5:54–65

Boothe O., Glassman D., 1987, "The Statistical Distribution of Exchange Rates", **Journal of International Economics**, Vol 22, pp 297-319

Branke J. (1995), "Evolutionary algorithms for neural network design and training", In Alander J. T., editor, *Proceedings First Nordic Workshop on Genetic Algorithms and their Applications*, pp 145–163, University of Vaasa Press, Vaasa, Finland.

Braun H. and Weisbrod J., 1993, "Evolving feedforward neural networks", In Albrecht R. F., Reeves C. R., and Steele N. C., editors, *Proceedings of ANNGA93, International Conference on Artificial Neural Networks and Genetic Algorithms*, pp 25–32, Springer-Verlag, Innsbruck.

Breiman L., 1996, "Bagging predictors", **Machine Learning**, 24, pp-123–140.

Chang W. and Mak M., 1999, "A Conjugate Gradient Learning Algorithm for Recurrent Neural Networks", **Neurocomputing**, 24, pp 173-189

Chen D. et al. (1993), "Constructive learning of recurrent neural networks", In Petsche T., Judd S.,and Hanson S., editors, *Computational Learning Theory and Natural Learning Systems III*. MIT Press, Cambridge, Massachusetts.

Chinn M. D., 1991, "Some Linear and Non-Linear Thoughts on Exchange Rates", **Journal of International Money and Finance**, Vol 10, pp 214-230

Elman J. L., 1990, *Finding structure in time*. **Cognitive Science**, 14:179-211

Fletcher R., 1987, *Practical Methods of Optimization*, Wiley-Interscience New York, NY, USA

Freund Y., Schapire R., "A decision-theoretic generalization of on-line learning and an application to boosting", In: Proceedings of the Second European Conference on Computational Learning Theory, Springer, Berlin, 1995, pp. 23–37.

Geman S., Bienenstock E., Doursat R., 1992, "Neural networks and the bias/variance dilemma", **Neural Computation**, 4, pp 1–58.

Goldberg D. E. and Richardson J., 1987, "Genetic algorithms with sharing for multimodal function optimization", In Grefenstette J. J., editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 148–154, Morgan Kaufmann, San Francisco, California.

Gomez, F. and Miikkulainen, R. 1999, "Solving non-Markovian control tasks with neuroevolution", In Dean, T., editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, ppp 1356–1361, Morgan Kaufmann, San Francisco, California.

Granitto P. M., Verdes P. F., Ceccatto H. A., 2005, "Neural Network Ensembles: Evaluation of Aggregation Algorithms", **Artificial Intelligence**, 163, pp139–162

Gruau F., Whitley D., and Pyeatt L. (1996), "A comparison between cellular encoding and direct encoding for genetic neural networks", In Koza J. R. et al., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 81–89, MIT Press, Cambridge, Massachusetts.

Hann T.H., Steurer E.. 1996, "Much Ado about Nothing? Exchange Rate Forecasting: Neural Networks vs Linear Models using Monthly and Weekly Data", **Neurocomputing**, Vol 10, pp 323-339

Haykin S., *Neural Networks:A Comprehensive Foundation*, Prentice Hall, Inc., *Upper Saddle River*, New Jersey

Hochreiter, Bengio Y., Frasconi P. and Schmidhuber J., '*Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies*'. In Kremer S. C. and Kolen J. F. (eds), A Field Guide to Dynamical Recurrent Neural Networks. IEEE press 2001

Hornik K., 1991, "Approximation Capabilities of Multilayer FeedForward Networks", **Neural Networks**, Vol 4, pp 251-257

Jaeger H. and Haas H., 2004, "Harnessing NonLinearity: Predicting Chaotic Systems and Saving Energy in Wireless Telecommunication", **Science**, April 2, pp 78-80

Jordan M. I., 1986, *Serial order: A parallel distributed processing approach*, Institute for Cognitive Science Report 8604. University of California, San Diego.

Kaelbling L. P., Littman M., and Moore A.W. (1996). "Reinforcement learning: A survey", **Journal of Artificial Intelligence**, 4:237–285.

Kosko B., *Neural Networks and Fuzzy Systems*, Englewood Cliffs, NJ, Prentice Hall, Inc. 1992

Krishnan R. and Ciesielski V. B., 1994, "Delta-gann: A new approach to training neural networks using genetic algorithms", In Tsoi A. C., and Downs T., editors, *Proceedings of the Australian Conference on Neural Networks*, pp 194–197, University of Queensland, Brisbane, Australia.

Lapedes A. and Farber R., 1988, "*How neural nets work*", In Anderson D. Z., editor, Neural Information Processing Systems, pp 442-456. American Institute of Physics, New York.

Lisi F., Schiavo R.A., 1999, "A Comparison between Neural Networks and Chaotic Models for Exchange Rate Prediction", **Computation Statistics and Data Analysis**, Vol 30, pp 87-102

Luenberger D.G., 1973, *Introduction to Linear and Nonlinear Programming*, Addison-Wesley Publishing Co., Reading, MA.

Mahfoud S.W., 1995, *Niching Methods for Genetic Algorithms* Ph.D. thesis, Department of General Engineering, IlliGAL Report 95001, University of Illinois at Urbana-Champaign, Urbana, Illinois.

Meese R. A., Rogoff K., 1983, "Empirical Exchange Rate Models of the seventies: Do They Fit Out of Sample?", **Journal of International** Economics, Vol 14, pp 3-24

Pineda F.J., 1988, "Generalization of backpropagation to recurrent and higher order neural networks", In Dana Z. Anderson (Ed.), *Neural Information Processing Systems*. New York: American Institute of Physics.

Refenes A. N.. 1996, "Constructive Learning and its Application to Currency Exchange Rate Forecasting, In *Neural Networks in*

Finance and Investing: Using Artificial Intelligence to Improve Real World Performance, Trippi RR, Turban E (eds). Probus Publishing Company: Chicago

Schmidhuber J., Wierstra D. and Gomez F., "Hybrid Neuroevolution/ Optimal Linear Search for Sequence Learning", Proceedings of the 19th International Joint Conference on Artificial Intelligence, (IJCAI), Edinburgh, p. 853-858, 2005.

Sharkey A.J.C., (Ed.), *Combining Artificial Neural Nets*, Springer, London, 1999.

Stanley Kenneth O. and Miikkulainen R., 2002, "Evolving Neural Networks through Augmenting Topologies", **Evolutionary Computation**, Vol. 10, Issue 2, pp 99-127

Weigend A.S., Hubermann B.A., Rumelhart D.E., 1992, "Predicting Sun Spots and Exchange Rates with Connectionist Networks", In *NonLinear Modelling and Forecasting,* Casdagli M., Eubank S. (eds). Addison-Wesley:Redwood City, CA; 395-432

Whiteson S., Stone P., Stanley K. O., Miikkulainen R., and Kohl N.. "Automatic Feature Selection in Neuroevolution." In GECCO 2005: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1225–1232, June 2005.

Williams R. J. and Zipser R. A., 1989, "A learning algorithm for continually training recurrent neural networks", **Neural Computation**, 1, 270--280.

Yadav R.N., Kalra P.K. and John J., 2006, "Time Series Prediction with Single Multiplicative Neuron Model", **Applied Soft Computing**

Yao X. and Liu Y., 1996, "Towards designing artificial neural networks by evolution", **Applied Mathematics and Computation**, 91(1):83–90

Zhang B. T., and M¨uhlenbein H., 1993, "Evolving optimal neural networks using genetic algorithms with Occam's razor", **Complex Systems**, 7:199–220.

Zhou Z. H., Wu J.,Tang W., 2002, "Ensembling neural networks: many could be better than all?", **Artificial Intelligence,** 137, pages 239–263.