THESIS FOR THE DEGREE OF MASTER OF SCIENCE

IMPROVING TIME SERIES PREDICTION USING RECURRENT NEURAL NETWORKS AND EVOLUTIONARY ALGORITHMS

ERIK HULTHÉN

Division of Mechatronics
Department of Machine and Vehicle Systems
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden, 2004

Improving Time Series Prediction using Recurrent Neural Networks and Evolutionary Algorithms

ERIK HULTHÉN

Division of Mechatronics
Department of Machine and Vehicle Systems
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31–772 1000

Chalmers Reproservice
Göteborg, Sweden, 2004

Improving Time Series Prediction using Recurrent Neural Networks and Evolutionary Algorithms

ERIK HULTHÉN
Division of Mechatronics
Department of Machine and Vehicle Systems
Chalmers University of Technology

Abstract

In this thesis, artificial neural networks (ANNs) are used for prediction of financial and macroeconomic time series. ANNs build internal models of the problem and are therefore suited for fields in which accurate mathematical models cannot be formed, e.g. meteorology and economics. Feedforward neural networks (FFNNs), often trained with backpropagation, constitute a common type of ANNs. However, FFNNs suffer from lack of short-term memory, i.e. they respond with the same output for a given input, regardless of earlier inputs. In addition, backpropagation only tunes the weights of the networks and does not generate an optimal design. In this thesis, recurrent neural networks (RNNs), trained with an evolutionary algorithm (EA) have been used instead. RNNs can have short-term memory and the EA has the advantage that it affects the architecture of the networks and not only the weights. However, the RNNs are often hard to train, i.e. the training algorithm tends to get stuck in local optima. In order to overcome this problem, a method is presented in which the initial population in the EA is an FFNN, pre-trained with backpropagation. During the evolution feedback connections are allowed, which will transform the FFNN to an RNN.

The RNNs obtained with both methods outperform both a predictor and the FFNN trained with backpropagation on several financial and macroeconomic time series. The improvement of the prediction error is small, but significant (a few per cent for the validation data set).

Key words: time series prediction, evolutionary algorithms, recurrent neural networks

Acknowledgements

I would like to thank my supervisor Dr. Mattias Wahde for his inspiration, support, and tempo. I am also grateful to Jimmy Pettersson for all his help throughout the course of this work.

Erik Hulthén
Göteborg, January, 2004

Table of Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Related work 2

1.3 Objectives 3

2 Methods **5**

2.1 Artificial neural networks 5

2.1.1 Feedforward neural networks 6

2.1.2 Recurrent Neural Networks 7

2.2 Training algorithms 10

2.2.1 Backpropagation 10

2.2.2 Evolutionary algorithms 12

3 Time series prediction **17**

3.1 Introduction 17

3.1.1 Difference series 17

3.1.2 Scaling 18

3.2 Benchmarks 19

3.2.1 Naive strategy 19

3.2.2 Exponential Smoothing 19

3.3 FFNNs trained with BP 19

3.4 RNNs trained with GA 21

3.4.1 Fitness 22

3.5 RNNs generated from FFNNs 22

4 Results **23**
4.1 Introduction 23
4.2 Time series 23
4.2.1 USD-JPY exchange rate 23
4.2.2 US unemployment rate 24
4.2.3 Dow Jones Industrial Average 25
4.3 Conclusion 26

APPENDED PAPER

Appended paper

This thesis contains the paper listed below. References to the paper will be made using the Roman numeral associated with the paper.

I. Erik Hulthén and Mattias Wahde, Improved time series prediction using evolutionary algorithms for the generation of feedback connections in neural networks, accepted for publication in *Proceedings of Computational Finance 2004*, Bologna, Italy, April 2004.

Chapter 1

Introduction

In this thesis recurrent neural networks, trained with evolutionary algorithms, are used for time series prediction. The results of the predictions are compared with results from feedforward neural networks, trained with backpropagation, and also with some other methods commonly used for time series prediction. This chapter introduces the subject and motivates the approach used in this thesis. Chapters 2 and 3 describe the methods used and time series prediction, respectively. In chapter 4 the results are reported and discussed.

1.1 Motivation

In time series prediction the task is to forecast the next value (values) in a data set. There are several fields in which time series prediction is of central importance, e.g. meteorology, geology, finance, and macroeconomics. Typically in those fields, there exists no accurate models of the system, and therefore the series are studied from a phenomenological, model-free point of view. In the physical sciences, where models are common, the use of model-free time series prediction is less common. Artificial neural networks (ANNs) are often used for time series prediction because of their ability to build their own internal models. A common method is to train feedforward neural networks (FFNNs) with backpropagation [5]. The method is easy to use and generally arrives quickly at small prediction errors. However, there are some drawbacks of using this method. First, in the training of an FFNN, by whatever method, one can never overcome the lack of short-term memory, illustrated in Fig. 1.1, and an FFNN is thus dependent on the number of lookback steps (further described in section 3.3). Second, backpropagation only tunes the weights in the FFNN and does not affect the design of the network. In order to achieve short-term memory a recurrent neural network (RNN), i.e. an ANN with feedback connections, can be used. An RNN can, in principle, store all former input signals and is thus not dependent on the number of lookback

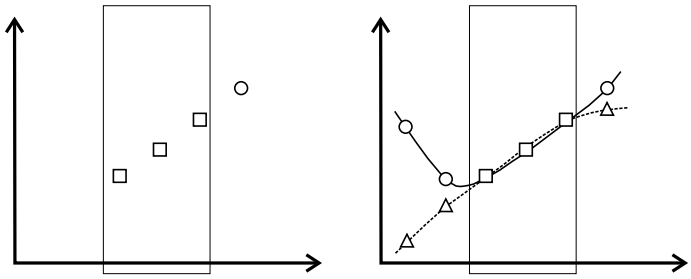

Fig. 1.1: An FFNN will always produce the same output for a given input.

steps. However, these networks cannot be trained with standard backpropagation. In this thesis evolutionary algorithms (EA) are used instead. Using an EA may also provide the advantage that the design of the networks becomes non-static, i.e. during the training not only the weights can be subject to change but also the architecture of the connections between the neurons and the size of the network.

Here a new method is presented, in which an FFNN is first trained with backpropagation, and then evolved further with an EA. The FFNN, duplicated and slightly mutated, forms the initial population of the EA. During the evolution feedback connections are allowed, i.e. the FFNN becomes an RNN if needed.

In order to illustrate the limitations of FFNNs a synthetic time series was generated, containing two situations with the same input values but different desired outputs, as shown in Fig. 1.1. The number of lookback steps was equal to 3 and the remaining 33 values were used for training and testing. The RMS distances between the real data set and the predictions were 0.0139, 0.0120, and 0.0080 for the FFNN, the RNN trained with random initial population, and the RNN trained with the FFNN as initial population, respectively. These results clearly illustrate the drawbacks caused by the lack of short-term memory in FFNNs.

1.2 Related work

ANNs have been used for time series prediction by several authors, e.g. [4], [3], and [7]. Many time series of interest, e.g. financial time series, have a high level of noise which is not always a result of insufficient measurements (which are often exact in the case of financial data) but the fact that the series come from systems with many diffuse influences, e.g. human psychology in financial series. On the other hand filtering the signal too much will take away small signs in the series that can give a hunch of the coming turns. In [4], the use of an RNN is preceded by preprocessing (differencing and log compression) and translation to a symbolic encoding with a self-organizing

map. The output from the RNN is used to build a deterministic finite state automaton describing some trend mechanisms. This procedure led to lower noise levels. FFNNs were used with success in [3] for prediction of USD-EUR exchange rate. The results were compared with other forecasting techniques using several comparison methods. In [7] the US Index of Industrial Production was forecast with an ANN that gave superior results compared to traditional methods.

Backpropagation has been used for training ANN in many situations, see e.g. [5]. Using an EA for evolving ANNs has been proposed by e.g. Yao [13]. Other ways to obtain ANNs with short-term memory is backpropagation through time (BPTT), described in [10]. In these networks each neuron has a connection to all neurons, N time steps back. However, the design of the network is still static and has to be specified beforehand, and one of the motivations for using ANNs is just its structural flexibility which is thus lost if BPTT is used. Another gradient-based method for training ANNs is real-time recurrent learning [11], but this method has the same drawbacks as BPTT.

1.3 Objectives

The objectives for this thesis are

- to investigate whether it is possible to obtain better time series prediction using RNNs trained with evolutionary algorithms instead of FFNNs trained with backpropagation.
- to test if an FFNN, trained with backpropagation, can be used successfully and efficiently to form an initial population in an EA that evolves RNNs for time series prediction.

Chapter 2

Methods

This chapter will introduce ANNs and their training algorithms, respectively. The time series used for prediction are transformed to data sets with input-output signals, and will be described in detail in chapter 3.

2.1 Artificial neural networks

ANNs are clusters of simple, non-linear function units called neurons, and are strongly inspired by biological neural networks found in animal brains. The history of artificial neural networks (ANNs) goes back to 1943 when McCulloch and Pitts presented a formula for artificial neurons that is basically the same as is used today [5]. The neurons are connected to each other via connections that transfer information. The connections, also called synapses, weigh the transferred signals. A neuron, shown in Fig. 2.1, consists of a summarizer and an activation function. The summarizer forms a sum of the input signals and a bias term, and gives the result to the activation function, which is the non-linear part of the neuron. One of its functions is to limit the output of the neuron to a given range, often [-1,1] or [0,1]. The output of the activation function is the output of the neuron itself.

Applications for ANNs can be found in various fields, e.g. the natural sciences, technology, and economics. A common application is image recognition, e.g. face recognition, automatic zip code reading, and analysis of satellite images [9]. ANNs can also be used as artificial brains in autonomous robots [8]. Another field in which ANNs are used is non-linear control. Time series prediction is also an important field, and constitutes the application in this thesis (described more in chapter 3).

There are two main reasons for the large computing power of ANNs [5]. First, their parallel distributed structure, which allows the networks to break down complex computation tasks into several easier ones, and second, the possibility to learn and make generalizations. The generalization makes it possible for an ANN to deliver a

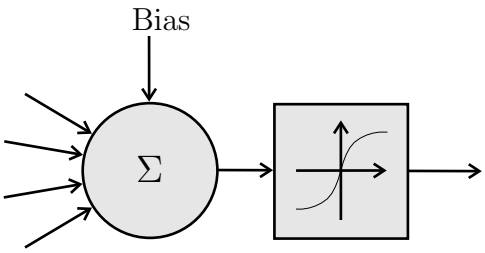

Fig. 2.1: The architecture of an artificial neuron: input signals, a bias term, a summarizer, an activation function, and an output signal.

reasonable output even if the given inputs were not part of the training set. This is also the reason for the robustness, i.e. insensitivity to noise, of ANNs. As an example, in face recognition, the image presented to the ANN as input is rarely exactly the same every time, but a well-trained ANN may still be able to give the right output. Another benefit of using ANNs is that they are model-free, i.e. it is not necessary to have a mathematical model of the system producing the inputs and outputs. This is especially important in application where the systems behind the data set are often difficult to model.

Here, a description of the two kinds of networks that have been used in this thesis will follow.

2.1.1 Feedforward neural networks

The feedforward neural networks, FFNNs, used here consist of input units and two layers of neurons as shown in Fig. 2.2. The number of input units and the number of neurons are constant for each data set. In this thesis, the number of output neurons is always equal to one. Each neuron has a connection to every neuron or input unit in the previous layer.

Neurons

Each neuron consists of a summarizer and an activation function. The summarizer collects the signals from earlier layers and adds a bias b_i. The activation function, σ, in all neurons is given by

$$\sigma(s) = \tanh(\beta s) \tag{2.1}$$

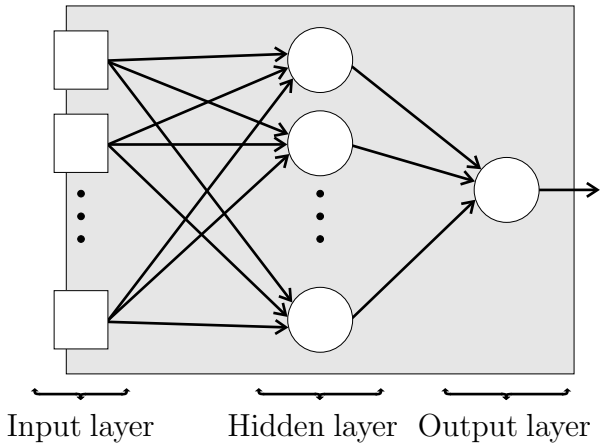

Fig. 2.2: A feedforward neural network. The circles are neurons, the boxes are input units, and the arrows are connection weights.

where β is a constant. Thus, the outputs from the neurons are computed as

$$y_i = \sigma\left(\sum_{j=1}^{n_x} w_{ij}x_j + b_i\right), i = 1, \ldots, n, \tag{2.2}$$

where n_x is the number of neurons or input units in the previous layer, x_j are the signals from that layer, and w_{ij} are the corresponding connections between the previous layer and the current layer. The bias term of neuron i, b_i, is used by the neuron to generate an output signal in the absence of input signals.

2.1.2 Recurrent Neural Networks

In contrast to FFNNs, RNNs may have connections (synapses) to all neurons in the network, i.e. there exists no neuron layers. This brings the fact that there is no obvious order for computing the output of the neurons. In biological neural networks, the problem with computing order does not exist. Instead, a biological neural network is a distributed processor allowing all neurons to execute independently. Because of its non-layered structure an RNN is more nature-like than an FFNN. In addition, RNNs have the possibility to maintain signals even after the input signals have disappeared. Thus, properly constructed RNNs have a short-term memory (see below).

Network equations

The RNNs in this thesis operate in continuous time. Signals from the neurons in RNNs are given by

$$\tau_i \dot{y_i} + y_i = \sigma\left(b_i + \sum_j w_{ij} y_j + \sum_j w_{ij}^{\mathrm{IN}} I_j\right), i = 1, \ldots, n \tag{2.3}$$

(see [8] and Paper I) where τ_i are time constants, σ the activation function, b_i bias terms, w_{ij} weights for the output signal y_j from the neuron j, and w_{ij}^{IN} weights for the external input signals I_j. $\dot{y}_i$ is approximated with Euler's method as

$$\dot{y}_i \approx \frac{y_i\left(t + \Delta t\right) - y_i\left(t\right)}{\Delta t}, \tag{2.4}$$

where Δt is the time step. The activation functions are given by

$$\sigma_i\left(s\right) = \tanh\left(\beta_i s\right), i = 1, \ldots, n, \tag{2.5}$$

where β_i are constants. The data sets used consist of input and output signals which are sampled with limited frequency. The time between two consecutive inputs is denoted ΔT and output signals are taken after each such period. The number of integration steps, N, between two output signals is

$$N = \frac{\Delta T}{\Delta t} \tag{2.6}$$

The external inputs are therefore constant for N steps, see below. Thus, the time discretized equation will be

$$y_i(t + \Delta t) = y_i(t) + \frac{\Delta t}{\tau}\left[\sigma\left(b_i + \sum_j w_{ij} y_j(t) + \sum_j w_{ij}^{\mathrm{IN}} I_j(t)\right) - y_i(t)\right] \tag{2.7}$$

The neuron outputs for the next $(N-1)$ time steps are calculated in the same way and the output at time $t + N\Delta t$ is taken as the next prediction from the network

$$\begin{aligned} y_i(t + N\Delta t) &= y_i(t + \Delta T) = \\ &= y_i(t + (N-1)\Delta t) + \\ &\quad + \frac{\Delta t}{\tau}\left[\sigma\left(b_i + \sum_j w_{ij} y_j(t + (N-1)\Delta t) + \sum_j w_{ij}^{\mathrm{IN}} I_j(t)\right) + \right. \\ &\quad \left. - y_i(t + (N-1)\Delta t)\right] \end{aligned} \tag{2.8}$$

The external input to the RNN remains constant for the N time steps.

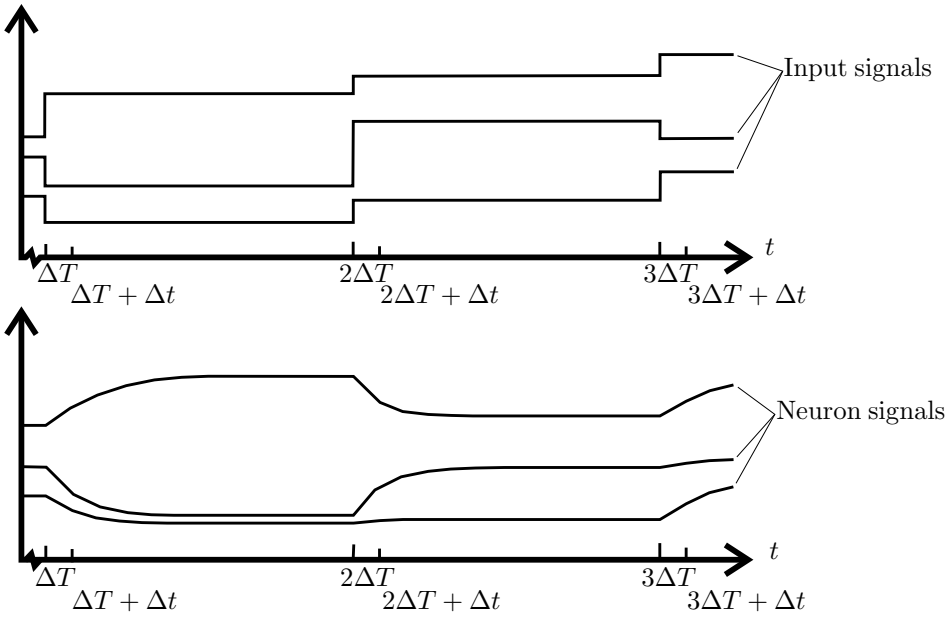

Fig. 2.3: Dynamical properties of an RNN.

Dynamical properties

As mentioned above, the RNNs used here operate in continuous time. Data points are, however, provided at discrete time steps. The external input to the RNN only changes when a new data point arrives. However, in the time period (of length ΔT) between two data points, the internal dynamics of the RNN unfolds according to Eq. 2.8. An example is shown in Fig. 2.3.

Note that an FFNN, as specified in section 2.1.1, is a special case of an RNN, namely one lacking feedback connections, and with all time constants τ_i approaching zero. In principle, neurons may be affected by earlier input signals (depending on time constraints). Thus, an RNN may contain a memory which is an advantage compared to the FFNN whose output is only affected by the current input signals.

Network elements

The RNNs consist of neurons and input units. As shown in Fig. 2.4, the connection weights may connect neurons both to input elements and neurons (note that autoconnections are allowed). The number of connections from an input unit or a neuron is set individually between zero and a maximum, M_{c}. An unconnected neuron has, of course, no effect on the output of the network.

The input units, which act as intermediaries between the input signals and the

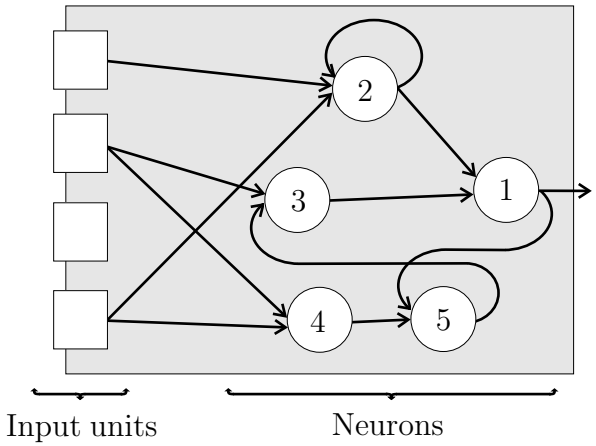

Fig. 2.4: A recurrent neural network. The arrows represent connection weights. The output of neuron 1 is taken as the output of the network.

neurons, distributing signals to one or several neurons, are updated every ΔT. In the applications considered here, only a scalar output is needed, and it is taken as the output from the first neuron in the RNN (see Fig. 2.4).

2.2 Training algorithms

The computation of the ANN is determined by a design process, known as training. For the training, backpropagation has been used for the FFNNs and evolutionary algorithms for the RNNs. Both methods will be described in this section.

2.2.1 Backpropagation

Backpropagation, see e.g. [9] and [5], is a gradient following method used to train FFNNs, i.e. to adjust the weights in the network. The weights between the inputs and the hidden layer are here denoted $w^{I \to H}$, and the weights between the hidden layer and the output layer are denoted $w^{H \to O}$. With an interval of ΔT (same as in section 2.1.2) the network provides an output upon the given input signals. This output, y^o, is compared to the correct value, o, for that time step giving the error, e, as follows

$$e = o - y^o \tag{2.9}$$

The weight modifications between the hidden layer and the output neuron are then given by

$$\Delta w_j^{H\to O} = \eta\delta y_j^H \tag{2.10}$$

where η is a learning parameter and δ the local gradient defined as

$$\delta = e\sigma'\left(\sum_{j=1}^{n^{(H)}} w_j^{H\to O} x_j + b_O\right) \tag{2.11}$$

where σ' is the derivative of the activation function. In a similar way the weight modifications between the input units and the neurons in the hidden layer are defined as

$$\Delta w_{ij}^{I\to H} = \eta\kappa_i y_j^I \tag{2.12}$$

where κ is a weighted sum of the δ-terms obtained from the output neurons (of which there is only one in the cases considered here), and is computed as

$$\kappa_i = \sigma'\left(\sum_{p=0}^{n^{(I)}} w_{ip}^{I\to H} y_p^I\right)\sum_{l=1}^{n^{(O)}} \delta_l w_{li}^{H\to O} \tag{2.13}$$

When weight modifications have been calculated for both the hidden layer and the output neuron, the new weights are computed as

$$w^{I\to H} \to w^{I\to H} + \Delta w^{I\to H} \tag{2.14}$$

and

$$w^{H\to O} \to w^{H\to O} + \Delta w^{H\to O} \tag{2.15}$$

When training the network, the time steps in the training set are considered in random order with weights updated after each output computation.

The learning parameter η can be either fixed or variable. Here, η is variable and defined as

$$\eta = \frac{\eta_0}{1 + \left(\frac{n_l}{\tau_\eta}\right)} \tag{2.16}$$

where n_l is the number of past training epochs, and τ_η and η_0 are constants. This way of calculating η is implemented by Haykin (but for a single neuron) in [5] and inspired by [1].

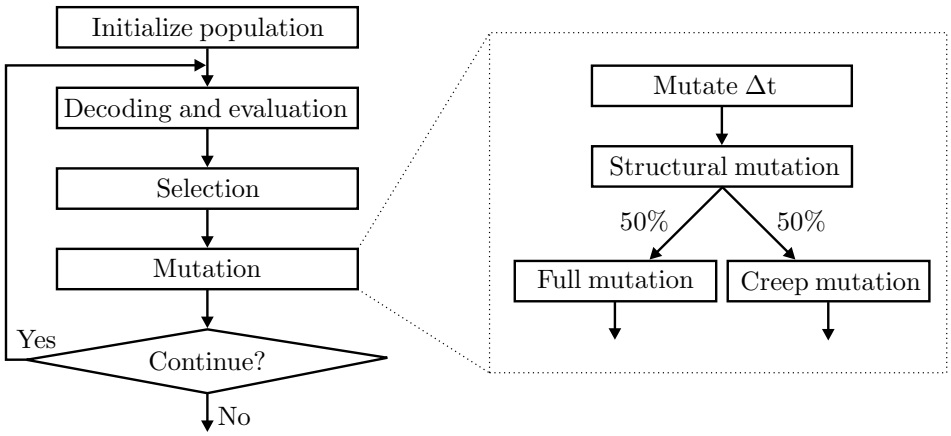

Fig. 2.5: Schematic representation of the implemented GA.

2.2.2 Evolutionary algorithms

Evolutionary algorithms is a general term for training methods inspired by darwinian evolution, and includes e.g. genetic algorithms (GA), genetic programming (GP), and Evolution strategies (ES) [9]. The methods have been used in many different applications e.g. optimization, construction of neural networks, time series prediction, control system identification, reverse engineering, and autonomous robot control. As the name implies, these methods are strongly inspired by natural evolution. Just as in nature, species form populations consisting of individuals. In nature individuals are living beings but in EAs the individuals are potential solutions to the problem under consideration. The variables of the problem are coded in the genomes of the individuals. A genome may consist of one or several chromosomes, which in turn consist of strings of values, known as genes. The solutions are tested and the individuals are given a measure of goodness, referred to as the fitness value. The higher the fitness, the larger the chance for an individual to be selected as parent to new individuals in the next generation. Reproduction may be sexual or asexual and mutations may occur. Although the process of an EA contains stochastic parts it should be stated clearly that evolution is not a random search. For a deeper review of this subject, [2] is recommended.

The EA used here is a modified genetic algorithm (GA). The main differences compared to a standard GA are that there is no crossover, structural mutations are added, and the mutation rate varies with the size of the network encoded in the chromosome (see Eq. 2.24 below). The crossover operator, which combines genes from two individuals, is rarely useful when evolving neural networks and is therefore not used here. In this thesis the genomes always consist of only one chromosome, but the number of

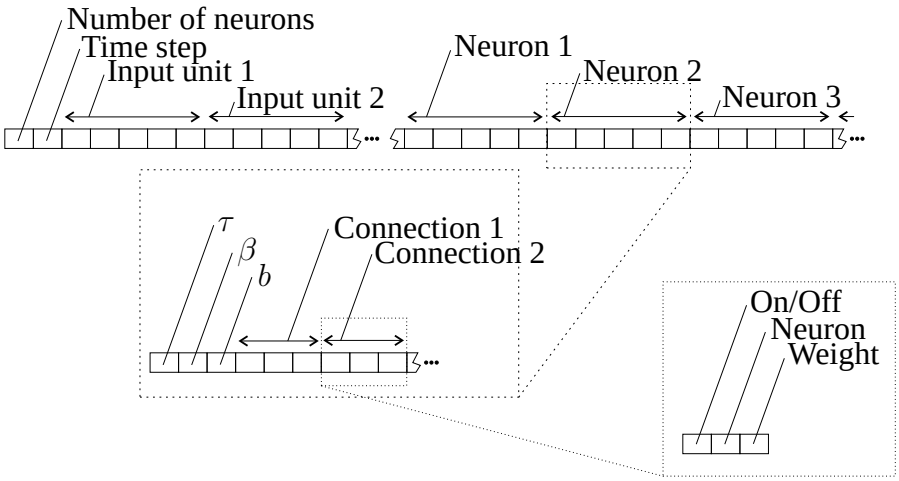

Fig. 2.6: The description of the RNN is encoded in a chromosome.

genes in a chromosome, and thus the number of neurons in the corresponding RNNs, may vary. The general progress of the GA used here can been seen in Fig. 2.5. The RNNs are then created from the decoded parameters of the chromosomes and tested in their application (in this case time series prediction). The test will give a fitness value back to the GA which is used for selection of parents used in the formation of the new generation. The selection method used here is tournament selection, described later in this section. The new generation is created asexually, meaning that the individuals (before mutation) are copies of their parents. The new generation is then exposed to mutations which will distinguish the genomes of the offsprings from those of their parents. When the new generation has been created a new evaluation takes place. The mutation is, as shown in Fig. 2.5, made in several steps. The structural mutations are essential since they let the RNN change size to adapt to the problem at hand. The mutation procedure will be described in detail below.

Encoding

The encoding of the chromosomes can be seen in Fig. 2.6. The chromosomes consist of genes represented by real numbers in the range $[-1, 1]$. The number of neurons, n, in the RNN is given by

$$n = |G_1|\, M_{\mathrm{n}} \tag{2.17}$$

where G_1 is the first gene in the chromosome G and M_{n} is the maximum number of neurons permitted. M_{n} is constant during the entire training. The time step Δt (as in

section 2.1.2) is given directly by

$$\Delta t = |G_2| \tag{2.18}$$

provided that $|G_2| > 0.001$, otherwise Δt is set to 0.001. Δt is initially set to a small value. Both neurons and input units are given by $3 + 3M_\mathrm{c}$ values in the chromosome, where M_c is the maximum number of connections from a neuron or input unit. However, the first three values are neuron-specific constants and are thus not used in the case of input units. The neuron-specific constants are τ, β, and the bias term b. In the following four equations G_j denotes those genes that encode the constant under consideration, see also Fig. 2.6. τ is given by

$$\tau_i = |G_j| \tag{2.19}$$

provided that $|G_j| > 0.001$, otherwise τ_i is set to 0.001.

$$\beta_i = C_\mathrm{s}\,(G_j + 1) \tag{2.20}$$

where C_s is a scale parameter, typically set to 2 here, thus letting β be in the range $[0, 2C_\mathrm{s}]$ and

$$b_i = C_\mathrm{t} G_\mathrm{j} \tag{2.21}$$

where C_t is a scale constant typically set to 1. Common for input units and neurons is that each possible connection is coded by three values: The first is an on-off flag (on if positive), the second denotes which neuron to connect to and the last is the weight of the connection. The index of the connected neuron is coded as follows

$$k = G_j M_\mathrm{n} \tag{2.22}$$

where M_n is the maximum number of neurons (same as in Eq. 2.17). The weight is coded according to Eq. 2.21 but with the scale constant C_t set to 2. The weights are thus in the range $[-2, 2]$. As is evident from Fig. 2.6, the length of the chromosome depends on the size of the encoded network.

Fitness function

For every individual a scalar fitness value is obtained that measures how well the individual performs. The selection of individuals is based on this value, i.e. the better the fitness, the larger the chance of being selected. The exact procedure by which the fitness is calculated in the problems considered here will be introduced in section 3.4.

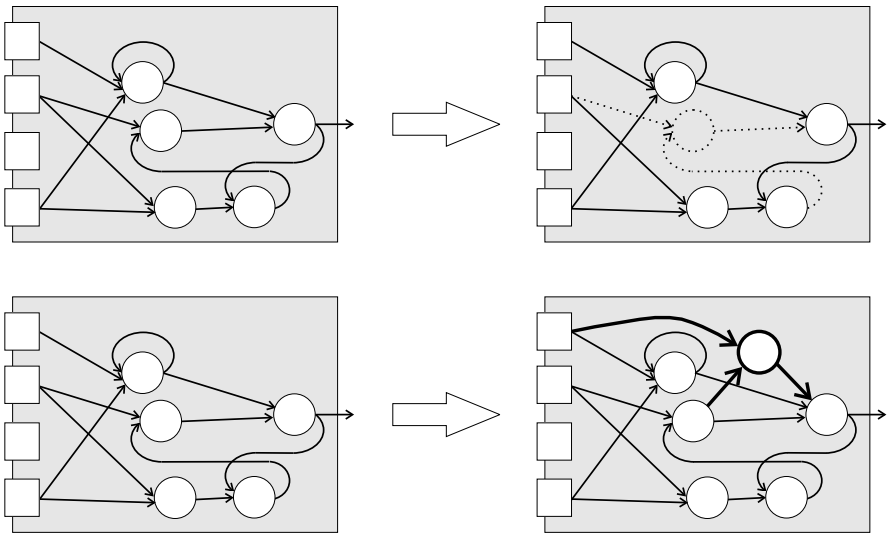

Fig. 2.7: Two types of structural mutations are used. In the top panel a neuron is removed. In the bottom panel a new neuron is added gently (i.e. using weak connection weights).

Tournament Selection

Tournament selection, used for selecting parents when forming new generations, operates as follows: a number, n_{T}, of different individuals are selected randomly from the population. n_{T} is calculated as

$$n_{\mathrm{T}} = \max(2, 0.15 n_{\mathrm{ind}}) \tag{2.23}$$

where n_{ind} is the size of the population. The individual to be copied to the next generation is with probability p_t the best of the n_{T} selected individuals, and with probability $1 - p_t$ any other individual, selected randomly. In this work, p_t is typically of the order 0.6 - 0.75. The selection is repeated n_{ind} times.

Mutation

The mutation rate is calculated for each individual as

$$p_{\mathrm{mut}} = \frac{10}{N_{\mathrm{G}}} \tag{2.24}$$

where N_{G} is the number of genes used in the RNN built from the corresponding chromosome. Not all genes are necessarily used (due to on-off flags in the chromosome and the variable number of neurons used). This formula was used with success in [8].

In each individual copied to the next generation either creep mutation or full scale mutation is made with equal probability. Creep mutation, which more gently modifies the individuals, is introduced as a complement to full scale mutation. All genes have the same probability to be mutated, whether they are used or not.

In full scale mutation one neuron or input unit is selected randomly. One of its genes, selected randomly, is then mutated. In addition, every other gene building that neuron or input unit is mutated with the probability p_{mut}. In full scale mutation, the new value of a mutated gene is selected randomly in the range [-1,1].

In creep mutation, as implemented here, only one gene in the chromosome mutates. The creep mutation is computed as

$$G_{\mathrm{new}} = G_{\mathrm{old}}\,(1 - 2rc + c) \tag{2.25}$$

where G is the gene, r is a random number in the range $[0, 1[$, and c is a constant creep rate, typically equal to 0.05.

The time step Δt mutates with probability p_{mut}. A time step mutation is performed as

$$\Delta t_{\mathrm{new}} = \frac{1}{\frac{1}{\Delta t_{\mathrm{old}}} \pm 1} \tag{2.26}$$

where Δt is the time step. Thus, for example, time step 0.2 may mutate to 0.25 or 0.1667.

Structural mutations have also been used, and the principles behind them are shown in Fig. 2.7. The mutation rates for structural mutations have been set empirically. With the rate $0.3p_{\mathrm{mut}}$, the network performs a structural mutation decreasing the number of neurons by one, provided that the network consists of more than one neuron. Outgoing connections are discarded and incoming connections are turned off. With mutation rate, $0.1p_{\mathrm{mut}}$, the network extends itself by adding a new neuron. The weights of the connection from the added neuron are limited to a small range with the purpose of avoiding a macro mutation by making a softer introduction of the new neuron.

Elitism

The best individual in each generation is copied to the next generation without any modifications. This guarantees that the maximum fitness of the population never decreases.

Chapter 3

Time series prediction

3.1 Introduction

A time series is a sequence of time-ordered values. In time series prediction, the task is, at a given time t, to estimate the value in the time series at time $t + f$. The input consists of the L latest values from the time series. In Fig. 3.1 this is shown with $L = 5$, and $f = 1$. For a series with m values, $m - L$ estimations ($\hat{x}$) can be made. In order to get an independent quality control, the data set is divided into a training set and a validation set [5]. The training algorithm only receives feedback from the training set, and the validation set is evaluated separately. When the training algorithm starts fitting noise (or other data behaviors not represented in the validation set) the error for the validation set will commonly stop decreasing and instead begin to increase. At this point, the training is terminated and the predictor that gives the lowest validation error is kept.

In order to compare the results from the different methods a root mean square prediction error is used, defined by

$$E = \sqrt{\frac{\sum_i^{m_c} (x_i - \hat{x}_i)^2}{m_c}} \tag{3.1}$$

where x_i are the correct values, $\hat{x}_i$ are the predictions, and m_c is the number of points compared.

3.1.1 Difference series

As an alternative to the original series, difference series are used in order to make the prediction easier. A difference series is created from the original according to

$$x_d(t) = x(t + 1) - x(t), t = 1, \ldots, (m - 1) \tag{3.2}$$

Difference series have been used in e.g. [3].

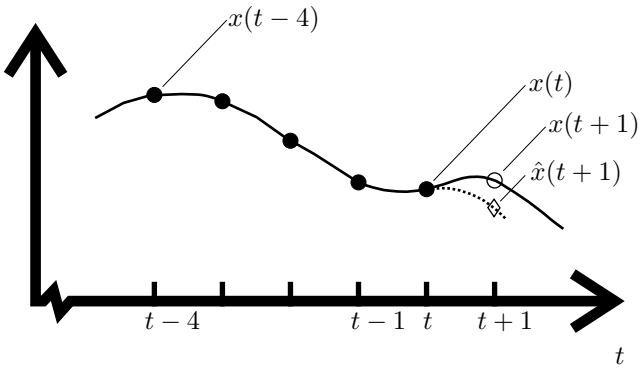

Fig. 3.1: Prediction of a time series for time $t+1$ given values at time $t...t-4$. The filled disks are the inputs and the open disk is the signal to which the prediction, $\hat{x}(t+1)$, is compared.

3.1.2 Scaling

In order to convert the input signals to the ANN to the range $[-1, 1]$ a scaling is performed according to

$$x_{\text{in}} = 2\frac{x - R_{\text{low}}}{R_{\text{high}} - R_{\text{low}}} - 1 \tag{3.3}$$

where x_{in} is the input signal to the ANN, x is the input signal as it is given before the scaling, R_{low} is the low range limit and R_{high} is the high range limit. The corresponding equation for the output signal from the ANN is given by

$$x = R_{\text{low}} + \frac{(R_{\text{high}} - R_{\text{low}})\,(x_{\text{out}} + 1)}{2} \tag{3.4}$$

where x_{out} is the output signal from the output neuron, R_{low} is the low range limit, R_{high} is the high range limit and x is the output in the original scale. The range limits are chosen to cover the range of the time series (R_{high} is typically rounded to the closest higher integer and R_{low} to the closest lower integer). In order to ensure that the output zero from the ANN also gives the prediction zero when rescaled, the scaling for the difference series is made with $R_{\text{high}} = -R_{\text{low}} = R$, where $R = \max(|x_{\text{d}}|)$, and x_{d} represents the elements in the difference series.

3.2 Benchmarks

In order to quantify the results from the neural networks some other methods were used as well.

3.2.1 Naive strategy

The naive strategy is the most simple strategy, simply saying: ”Tomorrow is like today” and is defined by $\hat{x}\,(t+1) = x\,(t)$. A useful predictor must at least surpass this naive method. In principle, the most simple network with only one connection from the latest input signal to one single neuron with the activation function $\sigma(s) = s$ would perform the naive strategy. The naive strategy is used as comparison method in e.g. [3] and [4]. The activation function used in this thesis gives zero as output when the input is zero. Thus, in the case of a difference series, a network with zero weights and biases already performs the naive strategy.

3.2.2 Exponential Smoothing

An alternative to the naive strategy, at least for smooth series, is linear exponential smoothing (LES), described e.g. in [6]. The forecast at the time t for f steps forward is given by:

$$\hat{x}(t+f) = S(t) + T(t)f \tag{3.5}$$

where S is the single exponentially smoothed value and T the trend. The single exponentially smoothed value is defined by

$$S(t) = \alpha_{\mathrm{S}} x(t) + (1-\alpha_{\mathrm{S}})\,(S(t-1) + T(t-1)) \tag{3.6}$$

where α_{S} is a constant in the range $[0,1]$ and x is the time series. The trend is a term added to include the current direction of the series and it is calculated as

$$T(t) = \alpha_{\mathrm{T}}\,(S(t) - S(t-1)) + (1-\alpha_{\mathrm{T}})\,T(t-1) \tag{3.7}$$

where α_{T} is a constant in the range $[0,1]$. The naive strategy described above is equal to LES when $f = 1$, $\alpha_{\mathrm{S}} = 1$, and $\alpha_{\mathrm{T}} = 0$.

3.3 FFNNs trained with BP

An FFNN that can be used for predicting e.g. the time series shown in Fig 3.1 is shown in Fig. 3.2. The FFNN is created with the same number of inputs as the number of lookback steps, L, in the time series. The input signals are presented to the network in such a way that input 1 always holds $x(t)$, input 2 holds $x(t-1)$ etc. The number of

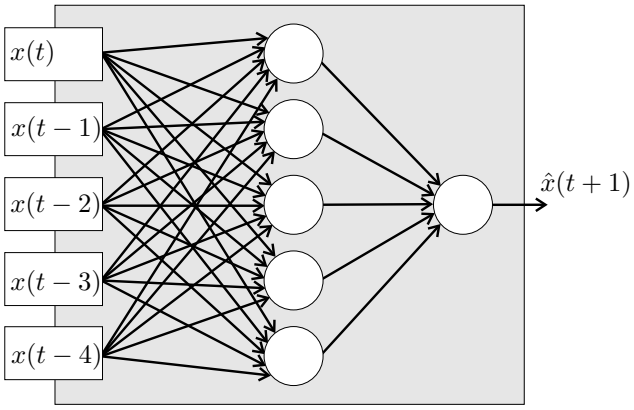

Fig. 3.2: An FFNN, with five neurons in the hidden layer, that estimates the time series x for time $t+1$ given signals from time $t-4$ to t.

neurons in the hidden layer, n_{h}, is determined empirically for each case, but is typically around 5. When n_{h} is chosen too large, the FFNN will overfit easily, i.e. the output from the network will be very good when the training data set is predicted but worse for the validation set, due to noise adaptation. On the other hand, a too small n_{h} will limit the capacity to a small number of different behaviors in the series. The constant β, which is the same for all neurons in the FFNN used here, is typically 1. Initially, the weights in the network are all set randomly in the range $[-1, 1]$. When the FFNN is trained, the weights are updated after each input-output pair (prediction). The input-output pairs are looped through in random order. The learning rate constant η_0 was typically about 0.02, while the learning time constant, τ_η, was set in the range $5 \cdot 10^4$ to $2 \cdot 10^6$.

However, as mentioned in chapter 1, the FFNN has no short-term memory. For instance, in the example shown in Fig. 1.1, with $L = 3$, the inputs are identical for the two series and the output from the FFNN will thus be identical irrespective of earlier input signals. This is a built-in limitation of FFNNs which cannot be evaded no matter what training algorithms are used. This problem can, of course, be solved by increasing L but then the number of weights will grow too (the number of weights is equal to $n_{\mathrm{h}}(L+2)+1$), and this will increase the risk of overfittning and make the training of the network more computationally demanding. Besides, increasing L will not solve the optimal design problem for the FFNN, i.e. the architecture of the network still has to be set beforehand. This is one of the main reasons for introducing the new method described in section 3.5.

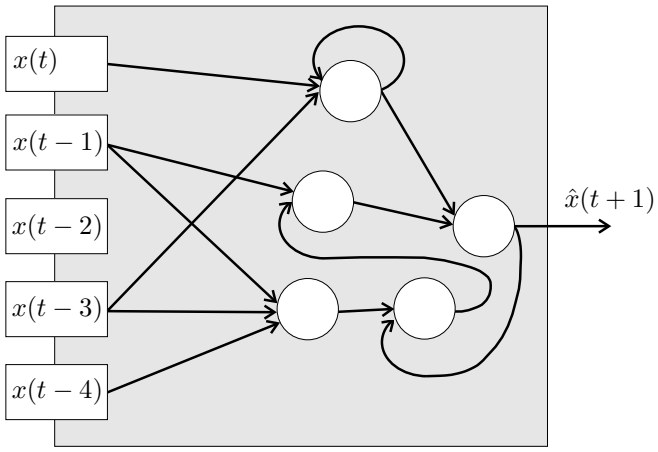

Fig. 3.3: An RNN that estimates the time series x at time $t+1$ given signals from time $t-4$ to t.

3.4 RNNs trained with GA

In Fig. 3.3, an RNN is to predict the time series x shown in Fig. 3.1. The number of input units is set to the number of lookback steps, L, in the time series. The input signals are presented to the network in consecutive order, i.e. input 1 always holds $x(t)$, input 2 holds $x(t-1)$ etc. Initially, the number of neurons is typically set to 2 but most often it will decrease to 1 when the training begins, before it starts to grow again. However, the number of neurons, the weights, and constants are fixed during the evaluation of a given RNN.

As mentioned in section 3.1, the training algorithm, in this case a GA, is only provided with feedback from the training data set, not the validation set. However, every network tried in the GA for the training data set is also tried for the validation data set outside the GA. This may have the effect that the best network on the validation set is a mutated individual which gave less good results for the training data set. It is not likely that this individual would be selected for reproduction, but if the individual performed best on the validation data set it is still recorded and the individual is kept for future reference.

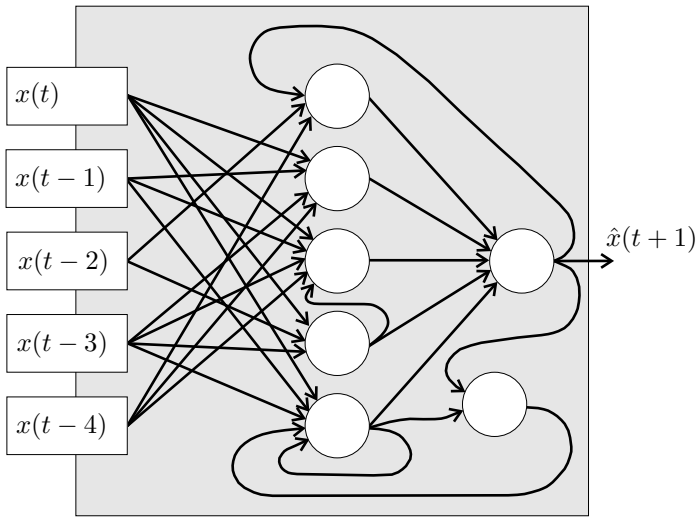

Fig. 3.4: An RNN, evolved from an FFNN, that estimates the time series x at time $t+1$ given signals from time $t-4$ to t. The likeness with the original FFNN, shown in Fig. 3.2, is evident.

3.4.1 Fitness

The fitness value, needed for the EA (described in section 2.2), is defined by

$$F = \frac{1}{E} \tag{3.8}$$

where E is the root mean square error calculated as in Eq. 3.1.

3.5 RNNs generated from FFNNs

In this method, proposed in Paper I, an ANN is first initialized and trained with back-propagation as described above in section 3.3. The weights are then coded into a chromosome (shown in Fig. 2.6) together with the constants: τ, β, and b. A population is formed from slightly mutated copies of this chromosome. Since the number of genes is typically $\sim 10^2$, mutation in too many genes would, with high probability, destroy the performance of the original FFNN, even in the case of creep mutations. A GA, as described above, but with this initial population instead of a random one, is then used for evolving RNNs. During the evolution, feedback connections are allowed if needed. An example is shown in Fig. 3.4. Compared to the FFNN in Fig. 3.2, new connections have been formed and some of the old connections have been removed.

Chapter 4

Results

4.1 Introduction

In this chapter, results from the analysis of three different time series are reported, namely the USD-JPY exchange rate, the US unemployment rate, and the Dow Jones Industrial Average.

The linear exponential smoothing method rarely gave better results than the naive method, i.e. the constants α_{S} and α_{T} could be set 1 and 0, respectively. For this reason, the results from the LES method are not shown separately in the result tables.

4.2 Time series

4.2.1 USD-JPY exchange rate

The data set used consisted of weekly averages of the daily interbank exchange rate from US dollar (USD) to Japanese yen (JPY) between 1986 and 2003. It contained 906 values, the unit of the values was Japanese yen, and the range was [82,164]. The averaging period eliminates weekly patterns. L was set to 4, which gave 902 values left for prediction, of which the first 539 were used for training and the remaining 363 for validation. For the difference series, the range was [-15,15] but the number of values used for training and validation was the same.

The RMS errors for both training and validation are shown in Table 4.1. The relative improvements from the naive strategy are shown in Table 4.2. The RNN obtained from the best FFNN in an EA, i.e. using the method proposed here, and in Paper I, gave the best results. Compared to the FFNN, during the evolution this RNN improved its results by 0.9% and 4.7% for the RMS validation and training error, respectively. The difference series in the exchange rate gave generally better results than the original series. Here, the RNN evolved from a random initial population performed best. How-

Table 4.1: Results from several different time series. The second column shows the lookback length and the subsequent columns show the error (normalized to [0,1]), obtained for the naive strategy, FFNN, RNN, and finally RNN evolved with FFNN as initial population. A 'd' after the series number implies that the difference series was used. Series I = USD - JPY interbank exchange rate, Series II = US Unemployment rate, and Series III = DJIA.

S.	L	Naive		FFNN		RNN		RNN f. FFNN	
		Tr.	Val.	Tr.	Val.	Tr.	Val.	Tr.	Val.
I	4	0.0183	0.0213	0.0191	0.0208	0.0216	0.0208	0.0182	0.0206
Id				0.0171	0.0207	0.0178	0.0203	0.0173	0.0204
II	5	0.0275	0.0163	0.0233	0.0161	0.0234	0.0154	0.0228	0.0159
IId				0.0229	0.0160	0.0251	0.0157	0.0243	0.0156
III	5	0.0704	0.0564						
IIId				0.0677	0.0535	0.0675	0.0530	0.0673	0.0530

ever, the RNN evolved from an FFNN made an improvement of 1.6% for the RMS validation error but suffered a decrease of 1.2% for the RMS training value.

According to [12], in which ANNs are used on several different currencies, it is difficult to forecast the trends in Japanese yen with technical algorithms. Their theory is that the market for JPY is large and more efficient, and therefore acts quickly on signs of change.

4.2.2 US unemployment rate

In this case the data set used consisted of monthly measurements of the US unemployment rate (seasonally adjusted), between January 1948 and September 2003. The series consisted of 669 values, the unit was per cent (%), and the range of the series was [2,11]. L was equal to 5, the training set consisted of the first 442 values and the validation set the remaining 222. For the difference series, the range was [-1.5,1.5] and the validation data set was one value shorter.

The RMS errors are shown in Table 4.1, and the relative improvements from the naive strategy are shown in Table 4.2. For the original series, the best result came from the RNN evolved from a random population. The other RNN, evolved from the FFNN, improved (lowered) the RMS error with 1.89% and 1.68% during the evolution for the training and validation data set respectively. For the difference series the best result was obtained from the RNN evolved from the FFNN. During the evolution, the validation RMS error sank by 2.12%, but rose by 6.51% for the training set. However,

Table 4.2: Improvements from the naive method used for each series. A 'd' after the series number refers to the difference series. Series I = USD - JPY interbank exchange rate, Series II = US Unemployment rate, and Series III = DJIA.

Series	FFNN		RNN		RNN from FFNN	
	Tr.	Val.	Tr.	Val.	Tr.	Val.
I	-3.92%	2.20%	-17.53%	1.93%	0.94%	3.12%
Id	6.90%	2.42%	2.77%	4.39%	5.79%	4.01%
II	15.52%	1.19%	14.95%	5.50%	17.11%	2.84%
IId	17.02%	2.04%	8.71%	4.09%	11.62%	4.12%
IIId	3.89%	5.08%	4.11%	6.01%	4.50%	5.89%

it should be noted that the FFNNs were only able to achieve a good fit for the training data set. Overall, the validation data set showed different behavior to the training data set, which can be seen in the results of the naive method. The RMS error for the training data set is almost 70% larger than the validation error.

The best RNN obtained from an FFNN is shown in Fig. 4.1.

4.2.3 Dow Jones Industrial Average

The series used is based on the average of 22 trading days (about a month) of daily Dow Jones Industrial Average (DJIA) closings from 1928 to 2003. Because of the increasing trend of the data set the series was transformed into a difference series according to

$$x_{\mathrm{d}}(t) = \frac{x(t+1) - x(t)}{x(t)}, t = 1, \ldots, (m-1) \tag{4.1}$$

The series consisted of 905 values in the range [-0.3,0.4]. L was equal to 5, 600 values were used for training, and 300 values were used for validation.

The RMS values are shown in Table 4.1, and the relative improvements from the naive strategy are shown in Table 4.2. The result from the naive method is calculated from the original series. The best result came from the RNN evolved from a random population. The other RNN, evolved from the FFNN, improved (lowered) the RMS error by 0.64% and 0.86% for the training and validation data sets, respectively.

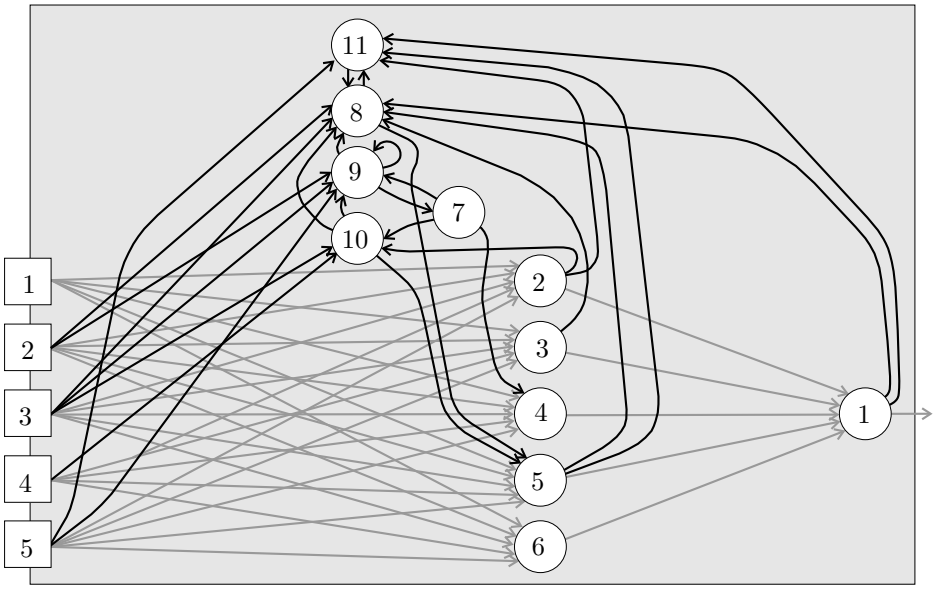

Fig. 4.1: The best RNN obtained from an FFNN for the US unemployment rate. The input units 1 to 5 and the neurons 1 to 6 descend from the original FFNN.

4.3 Conclusion

The general conclusion of this work is that it is possible to obtain better forecasts with an RNN than with an FFNN, but that the improvements are generally small. At best, improvements of a few per cent are obtained for the validation set and, for the training set, there is often a small degradation in performance compared to FFNNs.

Evolving an RNN from a random initial population is more difficult (and thus time-consuming) than training an FFNN using backpropagation. Furthermore, the evolutionary process does not always converge to a satisfactory network.

By contrast, the new method introduced in this thesis and in Paper I, in which RNNs are evolved starting from an initial population derived from an already trained FFNN, shows more reliable convergence and, in many cases, slightly better validation performance than the RNNs evolved from random initial populations.

Even though the improvements obtained with the new method are small in relative terms, they may be significant in absolute terms. As an example, on an average trading day, shares representing a market value of 38.5 billion USD are traded on the NYSE. Thus, a prediction difference of 0.1 per cent represents 38.5 million USD. Similarly, a 0.1 per cent difference in US unemployment represents a difference of 147,000 people.

Bibliography

[1] C. DARKEN, J. CHANG, AND J. MOODY, *Learning Rate Schedules for Faster Stochastic Gradient Search*, in Neural Networks for Signal Processing 2 - Proceedings of the 1992 IEEE Workshop, New Jersey, 1992, IEEE Press.

[2] R. DAWKINS, *The Selfish Gene*, Oxford University Press, New York, 1976.

[3] C. L. DUNIS AND M. WILLIAMS, *Modelling and Trading the EUR/USD Exchange Rate: Do Neural Network Models Perform Better?*, Derivatives Use, Trading and Regulation, 8 (2002), pp. 211–239.

[4] C. L. GILES, S. LAWRENCE, AND A. C. TSOI, *Noisy Time Series Prediction Using a Recurrent Neural Network and Grammatical Inference*, Machine Learning, 44 (2001), pp. 161–183.

[5] S. HAYKIN, *Neural Networks, A Comprehensive Foundation*, Prentice Hall, New Jersey, second edition, 1999.

[6] S. MAKRIDAKIS AND S. C. WHEELWRIGHT, *Forecasting Methods for Management*, Wiley, New York, fifth edition, 1989.

[7] J. E. MOODY, *Economic forecasting: Challenges and neural network solutions*, in Proceedings of the International Symposium on Artificial Neural Networks, Hsinchu, Taiwan, 1995.

[8] J. PETTERSSON AND M. WAHDE, *Generating balancing behavior using recurrent neural networks and biologically inspired computation methods*. Submitted to IEEE Transactions of Evolutionary Computation, September 2003.

[9] M. WAHDE, *An Introduction to Adaptive Algorithms and Intelligent Machines*, Göteborg, Sweden, second edition, 2002.

[10] P. WERBOS, *Backpropagation through time: what it does and how to do it*, Proceedings of the IEEE, 78 (1990), pp. 1550–1560.

[11] R. J. WILLIAMS AND D. ZIPSER, *A Learning Algorithm for Continually Running Fully Recurrent Neural Networks*, Neural Computation, 1 (1989), pp. 270–280.

[12] J. YAO, H.-L. POH, AND T. JASIC, *Foreign Exchange Rates Forecasting with Neural Networks*, September 1996, pp. 754–759.

[13] X. YAO, *Evolving Artificial Neural Networks*, Proceedings of the IEEE, 87 (1999), pp. 1423–1447.