Implementation and Evaluation of Two Prediction Techniques for the Lorenz Time Series

Grant E Huddlestone

Assignment presented in partial fulfilment of the requirements for the degree Master of Science at the Department of Applied Mathematics of the University of Stellenbosch, South Africa

Supervisor: Dr MF Maritz

March 2003

Declaration:

I, the undersigned, hereby declare that the work contained in this assignment is my own original work, and that I have not previously in its entirety, or in part, submitted it at any university for a degree.

Signature: .....

Date: 13/02/03

Abstract

This thesis implements and evaluates two prediction techniques used to forecast deterministic chaotic time series. For a large number of such techniques, the reconstruction of the phase space attractor associated with the time series is required.

Embedding is presented as the means of reconstructing the attractor from limited data. Methods for obtaining the minimal embedding dimension and optimal time delay from the false neighbour heuristic and average mutual information method are discussed.

The first prediction algorithm that is discussed is based on work by Sauer, which includes the implementation of the singular value decomposition on data obtained from the embedding of the time series being predicted.

The second prediction algorithm is based on neural networks. A specific architecture, suited to the prediction of deterministic chaotic time series, namely the time dependent neural network architecture is discussed and implemented. Adaptations to the back propagation training algorithm for use with the time dependent neural networks are also presented.

Both algorithms are evaluated by means of predictions made for the well-known Lorenz time series. Different embedding and algorithm-specific parameters are used to obtain predicted time series. Actual values corresponding to the predictions are obtained from Lorenz time series, which aid in evaluating the prediction accuracies. The predicted time series are evaluated in terms of two criteria, prediction accuracy and qualitative behavioural accuracy. Behavioural accuracy refers to the ability of the algorithm to simulate qualitative features of the time series being predicted.

It is shown that for both algorithms the choice of the embedding dimension greater than the minimum embedding dimension, obtained from the false neighbour heuristic, produces greater prediction accuracy.

For the neural network algorithm, values of the embedding dimension greater than the minimum embedding dimension satisfy the behavioural criterion adequately, as expected. Sauer's algorithm has the greatest behavioural accuracy for embedding dimensions smaller than the minimal embedding dimension. In terms of the time delay, it is shown that both algorithms have the greatest prediction accuracy for values of the time delay in a small interval around the optimal time delay.

The neural network algorithm is shown to have the greatest behavioural accuracy for time delay close to the optimal time delay and Sauer's algorithm has the best behavioural accuracy for small values of the time delay.

Matlab code is presented for both algorithms.

Opsomming

In hierdie tesis word twee voorspellings-tegnieke geskik vir voorspelling van deterministiese chaotiese tydreekse geïmplementeer en geëvalueer. Vir sulke tegnieke word die rekonstruksie van die aantrekker in fase-ruimte geassosieer met die tydreeks gewoonlik vereis.

Inbedmetodes word aangebied as 'n manier om die aantrekker te rekonstrueer uit beperkte data. Metodes om die minimum inbed-dimensie te bereken uit gemiddelde wedersydse inligting sowel as die optimale tydsvertraging te bereken uit vals-buurpunt-heuristiek, word bespreek.

Die eerste voorspellingsalgoritme wat bespreek word is gebaseer op 'n tegniek van Sauer. Hierdie algoritme maak gebruik van die implementering van singulierwaarde-ontbinding van die ingebedde tydreeks wat voorspel word.

Die tweede voorspellingsalgoritme is gebaseer op neurale netwerke. 'n Spesifieke netwerkargitektuur geskik vir deterministiese chaotiese tydreekse, naamlik die tydafhanklike neurale netwerk argitektuur word bespreek en geïmplementeer. 'n Modifikasie van die terugpropagerende leer-algoritme vir gebruik met die tydafhanklike neurale netwerk word ook aangebied.

Albei algoritmes word geëvalueer deur voorspellings te maak vir die bekende Lorenz tydreeks. Verskeie inbed parameters en ander algoritme-spesifieke parameters word gebruik om die voorspelling te maak. Die werklike waardes vanuit die Lorentz tydreeks word gebruik om die voorspellings te evalueer en om voorspellingsakkuraatheid te bepaal.

Die voorspelde tydreekse word geëvalueer op grond van twee kriteria, naamlik voorspellingsakkuraatheid, en kwalitatiewe gedragsakkuraatheid. Gedragsakkuraatheid verwys na die vermoë van die algoritme om die kwalitatiewe eienskappe van die tydreeks korrek te simuleer.

Daar word aangetoon dat vir beide algoritmes die keuse van inbed-dimensie groter as die minimum inbed-dimensie soos bereken uit die vals-buurpunt-heuristiek, groter akkuraatheid gee. Vir die neurale netwerk-algoritme gee 'n inbed-dimensie groter as die minimum inbed-dimensie ook beter gedragsakkuraatheid soos verwag. Vir Sauer se algoritme, egter, word beter gedragsakkuraatheid gevind vir 'n inbed-dimensie kleiner as die minimale inbed-dimensie.

In terme van tydsvertraging word dit aangetoon dat vir beide algoritmes die grootste voorspellingsakkuraatheid verkry word by tydvertragings in 'n interval rondom die optimale tydsvetraging.

Daar word ook aangetoon dat die neurale netwerk-algoritme die beste gedragsakkuraatheid gee vir tydsvertragings naby aan die optimale tydsvertraging, terwyl Sauer se algoritme beter gedragsakkuraatheid gee by kleiner waardes van die tydsvertraging.

Die Matlab kode van beide algoritmes word ook aangebied.

Contents

List of Figures

List of Tables

List of Symbols

T: The Lorenz time series used for algorithm evaluation.

N: The length of the time series T.

x_t: A discrete variable describing the state of the system at time t.

m: The embedding dimension.

l: The time delay.

s: The spanning dimension.

$\mathbf{b}_t$: The prediction point on the reconstructed Lorenz attractor.

$\mathbf{n}_i$: The i-th nearest neighbour to the prediction point on the Lorenz attractor.

e_i: The target point for the i-th nearest neighbour.

$\mathbf{c}$: The centre of mass of the nearest neighbours.

$\mathbf{d}(t)$: A time delayed coordinate set at time t.

k: Regression coefficients.

h_i^n: The summing sub-unit value for node i in layer n.

L_i^n: The value for node i in layer n.

w_{ij}^n: The weight connecting the j-th node in layer $n-1$ with the i-th node in layer n.

$\Psi(p)$: Input pattern p for a neural network.

$\Upsilon(p)$: Output pattern p for a neural network.

Λ: The number of layers in a neural network.

H: The number of context layers in a neural network.

J: The number of sets of input patterns, $\Psi(p)$, and associated output patterns, $\Upsilon(p)$, for a neural network.

M^n: The number of nodes in layer n.

Chapter 1

Introduction

1.1 Background

When a discussion turns to prediction, a person will invariably ask whether predicting the financial markets is possible. This question highlights the common desire to divine the future and has been with humankind for millennia, at first simply wondering what the weather might hold for crops to the modern day desire to forecast the market-driven economies of the world.

In explaining the process of prediction to a person, the distinction between different forms of chaotic behaviour has to be highlighted. Purely stochastic behaviour is what people immediately think about when they hear chaos. The term chaos is however also considered for systems that illustrate irregular behaviour but at the same time are not completely random.

This form of chaos is deterministic [1] [12] and has two key features; the system is sensitive to initial conditions and the system, over time, settles down to a fixed region of the space defined by its variables.

Sensitivity to initial conditions means that a very small change in the initial, or starting, conditions of a system can lead to very different behaviour in a short period of time. For the purposes of this thesis, only a deterministic chaotic system that possesses an attractor, being a localized area in the variable space for the system to which the solution will converge from initial conditions, will be considered.

Accurately predicting the future for a stochastic system might to all intents and purposes be impossible. This however is not the case for a system exhibiting deterministic chaotic behaviour where methods may be sought to make predictions [11] [27] about the future behaviour of the system.

As with all analytical methods the data under investigation, in this sense the time series containing observations of the chaotic system, has to be processed into a form that is convenient for analysis.

A large number of prediction algorithms [3] [4] [8] [18] [21] rely on the reconstruction of the attractor from the observed measurements. Due to the potentially large number of variables at play in a chaotic system,

reconstruction of the attractor is usually not possible and a method is sought to approximate the attractor. Sauer [20] justified the use of embedding a time series in reconstructing an attractor. Successful embedding requires the choice of a minimal embedding dimension [14] and an optimal time delay [9]. Methods exist to determine these optimal parameters, and these methods will be discussed in chapter 2.

1.2 Problem Description

This thesis implements and evaluates the performance of two prediction algorithms when applied to the Lorenz time series (section 1.3).

The first algorithm is based on work by Sauer [18] and utilizes local linear models, in which the singular value decomposition plays a pivotal role, to perform the predictions.

The second algorithm employs a neural network approach, which, because of the non-linear nature of the method has appeared suitable for the analysis and prediction of chaotic data. Quoting Swingler [25]: "*applications which require pattern recognition or simulation of a physical system too complex to model with rules are perfectly suited to neural computing techniques.*" Various prediction algorithms [2] [3] [4] [7] [17] based on neural networks have been formulated.

The performance of these two algorithms is evaluated on the basis of (1) prediction accuracy and (2) whether the predicted time series captures the dynamics of the time series under investigation.

The first criterion of prediction accuracy as a measure of performance is the prime purpose of the algorithm. However, the second criterion asks a more stringent question: if the algorithm predicts a number of time series values into the future, do those values imitate the behaviour of the time series being predicted?

The question then arises about which combinations of embedding dimension and time lag are the best for prediction accuracy and for capturing of the dynamics. Furthermore, are the two criteria satisfied by the same combination of test parameters? Finally, are the best combinations independent of the choice of the algorithm?

1.3 Lorenz system, general information and equations

The Lorenz time series [16] [23] is a staple source of data for the evaluation of prediction algorithms.

The Lorenz equations describe a simplified model of fluid motion driven by convection in the atmosphere by incorporating variables for the rate of convection overturning (x), the horizontal temperature variation (y), and the vertical temperature variation (z) and is given by,

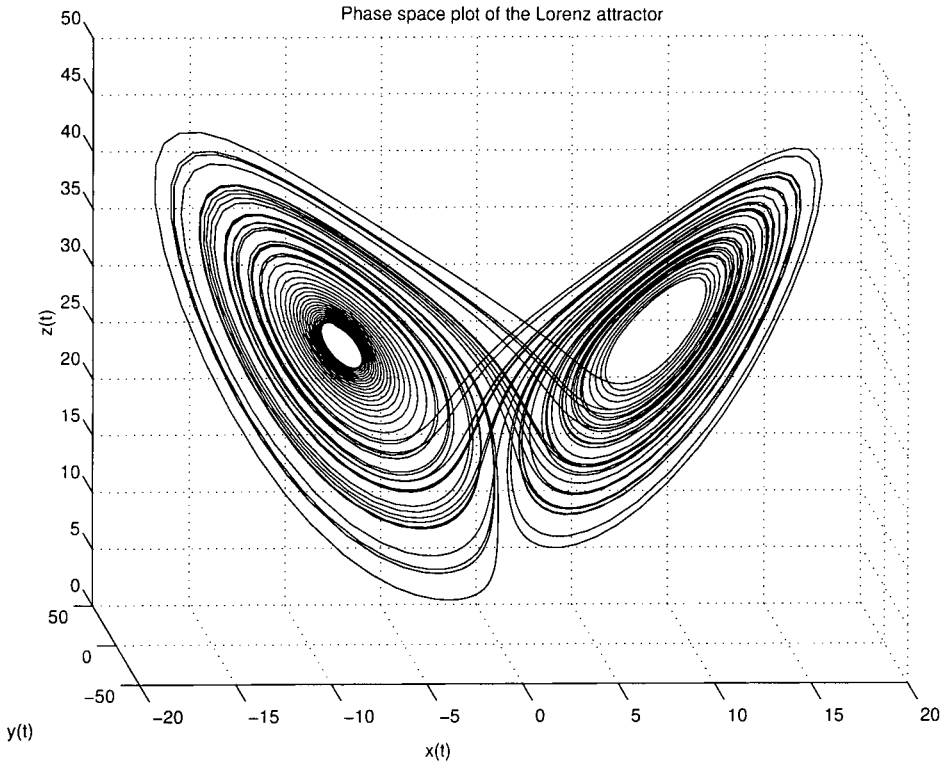

Figure 1.1: Phase space plot of the Lorenz attractor obtained from the solution of the three Lorenz equations.

$$\frac{dx}{dt} = \sigma(y - x), \tag{1.1}$$

$$\frac{dy}{dt} = \rho x - y - xz, \tag{1.2}$$

$$\frac{dz}{dt} = -\beta z + xy. \tag{1.3}$$

The Prandtl number, σ, is proportional to ratio of the fluid viscosity of a substance to its thermal conductivity. The Rayleigh number, ρ, is proportional to difference in temperature between the top and bottom of the system and β is the ratio of the width to the height of the Rayleigh-Bénard cell encasing the system. These parameters are usually set to the following values; $\sigma = 10$, $\rho = 28$ and $\beta = \frac{8}{3}$.

The Lorenz equations have certain important features:

- The equations are autonomous, i.e. the right-hand sides are not functions of time.
- The equations are non-linear because of the xz and xy terms.
- The solutions are bounded.
- There are no unstable periodic orbits or unstable stationary points.

Even though the equations contain no random, noisy or stochastic terms there is a surprising level of unpredictability in the solution, as is evident from figure 1.1.

The choice of the Lorenz equations to test prediction algorithms is motivated by:

- The infinite number of data points that can readily be generated computationally from the equations.

- The solution of the Lorenz equations describes a strange attractor in phase space.

The solution for equations (1.1), (1.2) and (1.3) were obtained from the utilisation of an explicit Runge-Kutta formula. In this work the ode45 function of Matlab was used.

Solutions can be found for x, y and z but this thesis will only utilise the time series obtained from the solution of y.

Let $x_i = y((i-1)h)$, where h is the time step for the solution obtained from the function solver and for this paper is set at 0.0170s. Therefore, the time series used for algorithm evaluation in this thesis has the form,

$$T = \{x_1, x_2, \ldots, x_n\}, \tag{1.4}$$

and is graphically represented in figure 1.2.

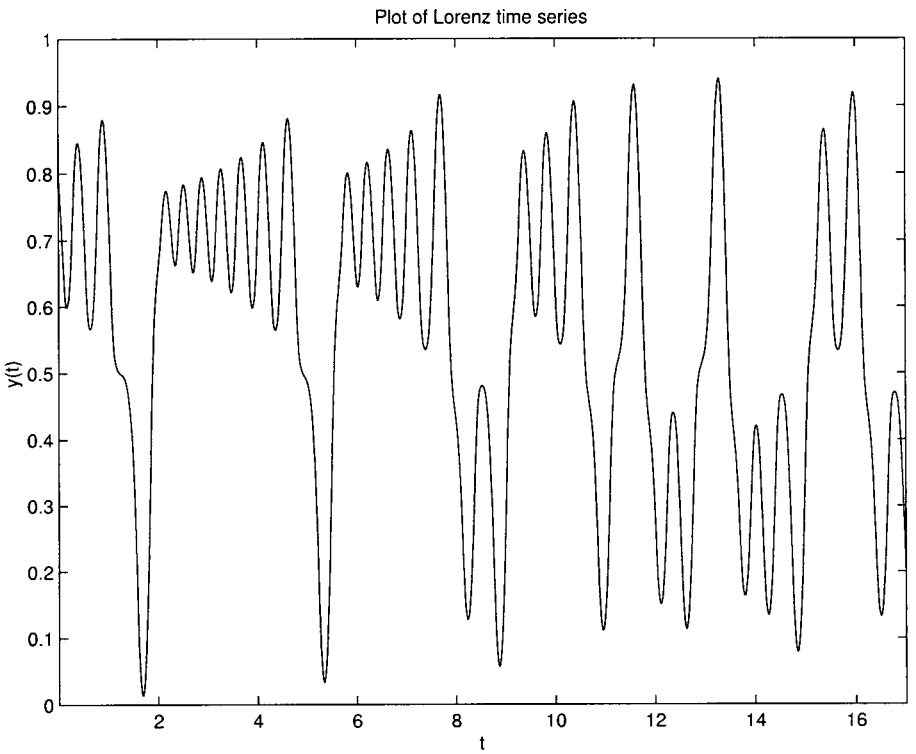

Figure 1.2: Plot of the solution of y(t) for the Lorenz systems of equations.

1.4 Notation conventions

Vectors are printed in boldface and denote a column vector. The transpose of vectors will be indicated by the superscript $^{\mathsf{T}}$. The Euclidean norm of a vector $\mathbf{a} = [a_1, a_2, \ldots, a_n]^{\mathsf{T}}$ will be represented by $\|\mathbf{a}\|$ with

$$\|\mathbf{a}\| = \sqrt{\mathbf{a} \cdot \mathbf{a}} = \sqrt{\sum_j a_j^2}. \tag{1.5}$$

1.5 Chapter Preview

Chapter 2 explains the fundamental ideas behind embedding theory and elaborates on the processes behind calculating the optimal embedding dimension and time delay for a given time series.

Having laid the groundwork for prediction algorithms to come, chapter 3 explains Sauer's algorithm. Chapter 4 briefly introduces the reader to main concepts in neural network architecture as well as explaining the time dependent architecture utilized in the algorithm.

The implementation of the algorithms is explained in chapter 5 and results and analysis of both algorithms are presented.

Chapter 2

Concepts in embedding theory

A dynamical system is a set of equations that gives the time evolution of the system's state from initial conditions.

The dynamical system may either be described by continuous variables, where a common form of the dynamical system can be described by first-order autonomous ordinary differential equations such as those for the Lorenz equations (1.1, 1.2, 1.3). The solution for the differential equations describes a continuous evolution of the state of the system from initial conditions.

Similarly, for discrete variables, the dynamical system may be described by a map of the form,

$$\mathbf{x}_{n+1} = \mathbf{F}(\mathbf{x}_n), \tag{2.1}$$

for which the solution describes a discrete evolution of the state of the system from initial conditions.

Each multi-dimensional point obtained from the solution of the dynamical system describing the state of the system is represented by a corresponding point in phase space whose coordinates are the variables of the state.

Therefore any point in phase space completely describes the state of the system and conversely any state of the system has a corresponding point in phase space. The relationship (interconnection) of points in phase space describing the time evolution of the system is termed a trajectory.

If, for some initial conditions the trajectories converge to some bounded subset of phase space, this set is referred to as an attractor. Transient data are values obtained for the system that have not yet converged onto the attractor. Due to fact that transient behaviour does not capture the stable dynamics of the system, its effect on prediction accuracy is an important consideration and thus for the remainder of this thesis it is assumed that all transient behaviour is omitted from the relevant data sets.

If the variables observed from a system do not fully describe a phase space point then some method is required to reconstruct phase space from the incomplete set of variables.

2.1 Phase space reconstruction

Reconstruction of the original phase space from observed data is usually not possible. Methods are thus sought which allow the reconstruction of the attractor in phase space from the available data (for earlier methods see [10] and [19]). The requirement for reconstruction is that linearisation of the dynamics at any point in the phase space is preserved by the reconstruction process. This is due to the need to exploit determinism in the data and the provision that the time evolution of a trajectory in reconstructed phase space only depends on its current position.

Given an observed discrete time series, T, a delay reconstruction, with dimension m and time delay l, is as follows,

$$\mathbf{b}_{(m-1)l+n} = \left[x_{(m-1)l+n}, \ldots, x_{2l+n}, x_{l+n}, x_n\right]^T. \tag{2.2}$$

Embedding theory states that for ideal noise-free data there exists a minimal value for the dimension, m, such that the reconstructed vectors $\mathbf{b}$ are equivalent to phase space vectors. Equivalence, in this sense, refers to the fact that the map is a one-to-one immersion (to be explained below).

A construction is only considered an embedding if the reconstructed map does not collapse points or tangent directions, in other words, that the mapping is one-to-one and that differential information is preserved.

When the state of the deterministic dynamical system, and the future evolution, is completely specified by a single point in phase space then the mapping is considered one-to-one. If at a given state x the value $f(x)$ is observed in the reconstructed space and that after a fixed time interval another event follows, then if f is one-to-one each appearance of $f(x)$ will be followed by the same event.

If a smooth map, F on A, is a continuously differentiable manifold and the derivative map $DF(x)$ is one-to-one for every point x on A then the map is an immersion. Under an immersion no differential structure is lost in going from A to F(A).

Therefore a map, F, is an embedding if and only if it is a one-to-one immersion. Figures 2.1, 2.2 and 2.3 illustrate different valid and invalid embedding maps.

In order to embed a scalar time series we need to construct m independent variables, the number required to uniquely characterize the system. The variable set (vector) will thus exist in some m-dimensional space. Therefore, to reconstruct the attractor in the new phase space we need to find the embedding dimension, m, and an optimal time delay, which will allow the best choice of values from the time series so that the variables are independent.

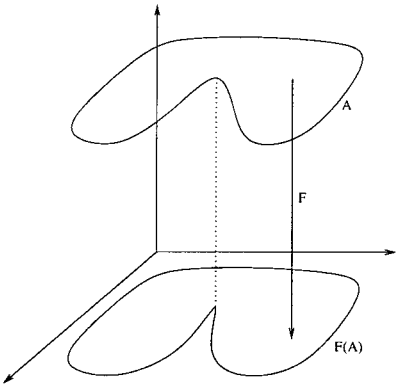

Figure 2.1: A one-to-one map that is not continuously differentiable, i.e. not an immersion.

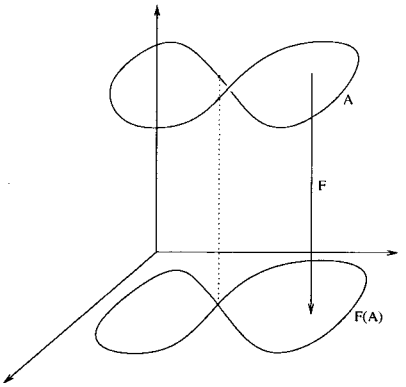

Figure 2.2: A map that is not one-to-one.

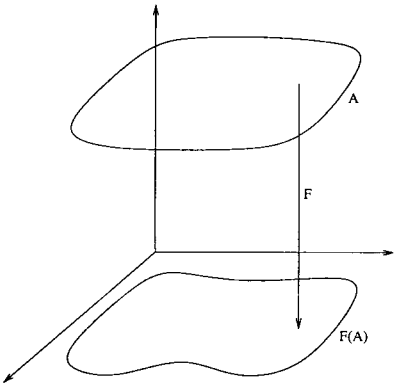

Figure 2.3: An embedding F of the smooth map A.

2.2 Time Delay

The reconstruction of phase space is equivalent to the original space of variables and is a method for providing independent coordinates composed of the present observation x_t and earlier observations of the system. The time delayed coordinate set at time t, with time delay l, appears as follows

$$\mathbf{d}(t) = [x_t, x_{t-l}, x_{t-2l}, \ldots,]. \tag{2.3}$$

If the time delay is chosen too small then successive delayed coordinates will not be independent enough and not enough time would have passed for the system to have explored the state space sufficiently. As an example, consider a highly oversampled data set where given a small time delay successive time delayed measurements would be almost identical and provide no information about the system.

Conversely, if the time delay were too large, two successive measurements in equation 2.3 would be nearly random with respect to each other.

Figure 2.4 illustrates the choice of time delays for the Lorenz time series. It can be seen that for a time delay

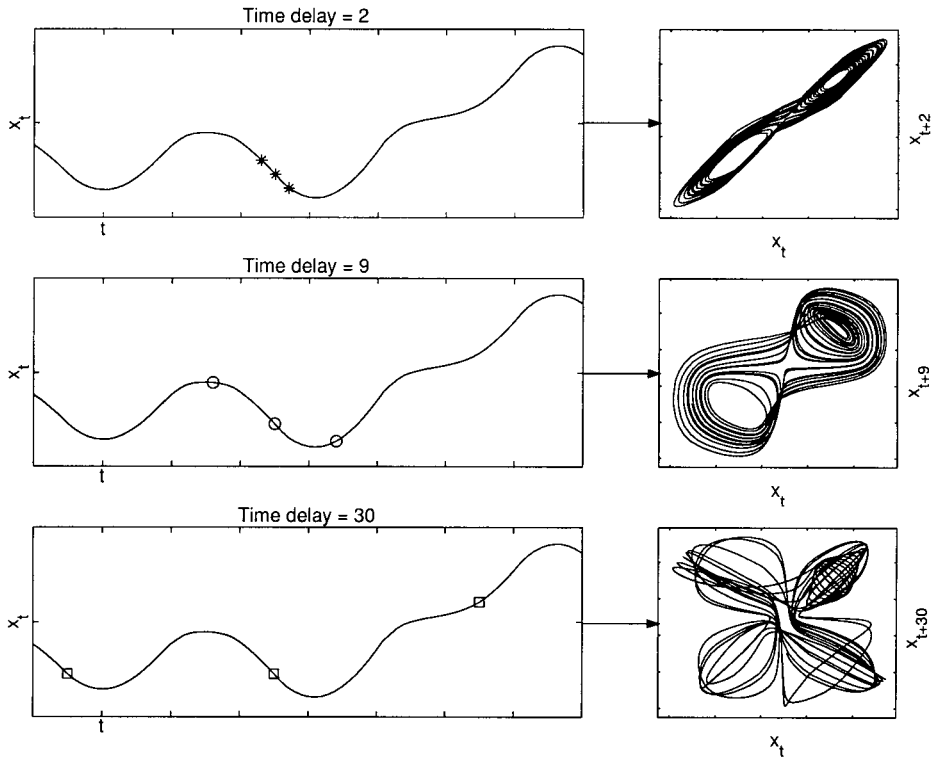

Figure 2.4: Different choices of time delays on a Lorenz time series.

of 2 very little information about the evolution of the system is obtained and that for a time delay of 30 the points have a near random distribution and likewise give no information on the state of the system. For a time delay of 9 it can be felt by visual inspection that the dynamics of the system is being captured. This value for the time delay will be shown to be the theoretical correct choice for the Lorenz time series. Figure 2.4 also shows the Lorenz attractor in 2 dimensions for the different time delays. It can once again be seen that incorrect choices of the time delay lead to bad reconstructions of the attractor.

A value for the time delay is thus sought, which will ensure that successive time delayed measurements, are not completely independent in a statistical sense. A method is sought that quantifies the dependence of $\mathbf{d}(t-l)$ on values of $\mathbf{d}(t)$ in a time series.

Mutual information is defined by

$$I(l) = -\sum_{ij} p_{ij}(l) \ln \frac{p_{ij}(l)}{p_i p_j},$$

where p_i is the probability to find an observable in the i-th interval of a time series and p_{ij} is the probability that an observation falls into the i-th interval and after a time l is observed in the j-th interval. Fraser [9] demonstrates that the first minimum of the mutual information of a time series corresponds to the best choice of time delay.

In figure 2.5 the average mutual information for the Lorenz time series is shown with the first minima indicating

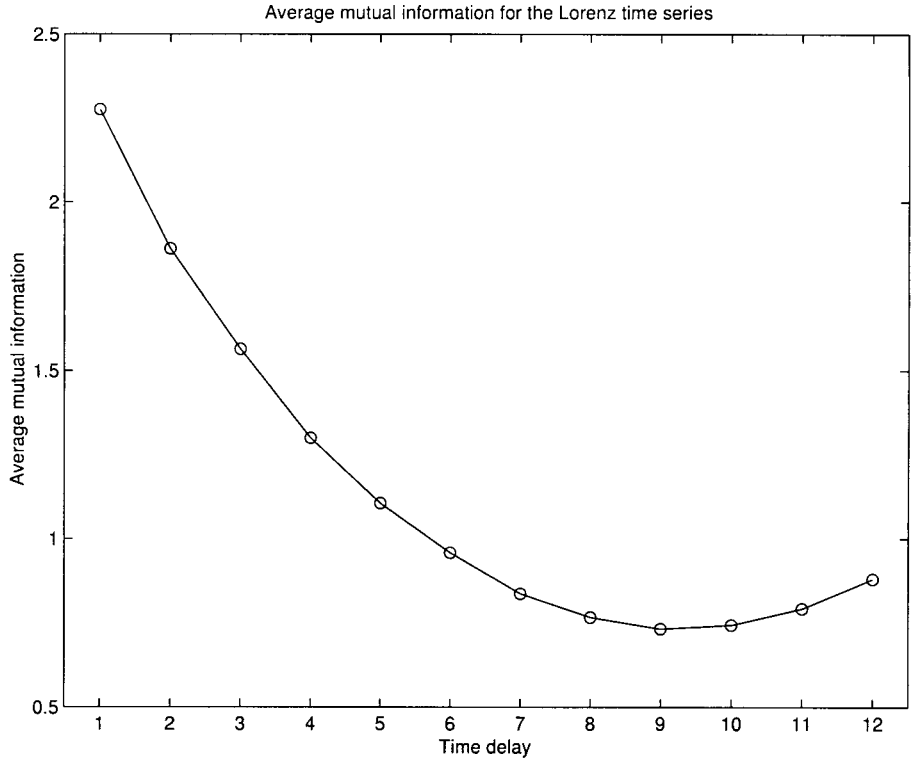

Figure 2.5: Average mutual information versus time delay for the Lorenz time series.

the optimum time delay at 9. Experimental results in chapter 5 illustrate how the choice of time delay affects prediction accuracy.

2.3 Embedding Dimension

An attractor is considered unfolded, and thus an embedding, if all neighbouring points in the attractor are true neighbours. Kennel [14] defines a false nearest neighbour as follows: *"A false neighbour is a point in the data set that is a neighbour solely because we are viewing the orbit (the attractor) in too small an embedding space."* Therefore all neighbouring points, in an incomplete unfolded attractor, will not be so solely due to the dynamics of the system but also because the geometric structure of the attractor was projected down to smaller space than required by the attractor. It is obvious that predictions made with information from false neighbours would deliver unreliable results.

The method of false nearest neighbours, introduced by Kennel [14], is used to calculate a minimum embedding dimension for the attractor so as to avoid projecting points into neighbourhoods of the attractor to which they do not belong. This means that for dimensions m less than the minimum embedding dimension m_0 the topological structure of the attractor is not preserved and the reconstruction is not a one-to-one image. Therefore, once the number of false nearest neighbours is zero the minimum embedding dimension has been

reached and the attractor has been unfolded in phase space. Figure 2.6 illustrates the effect of false neighbours.

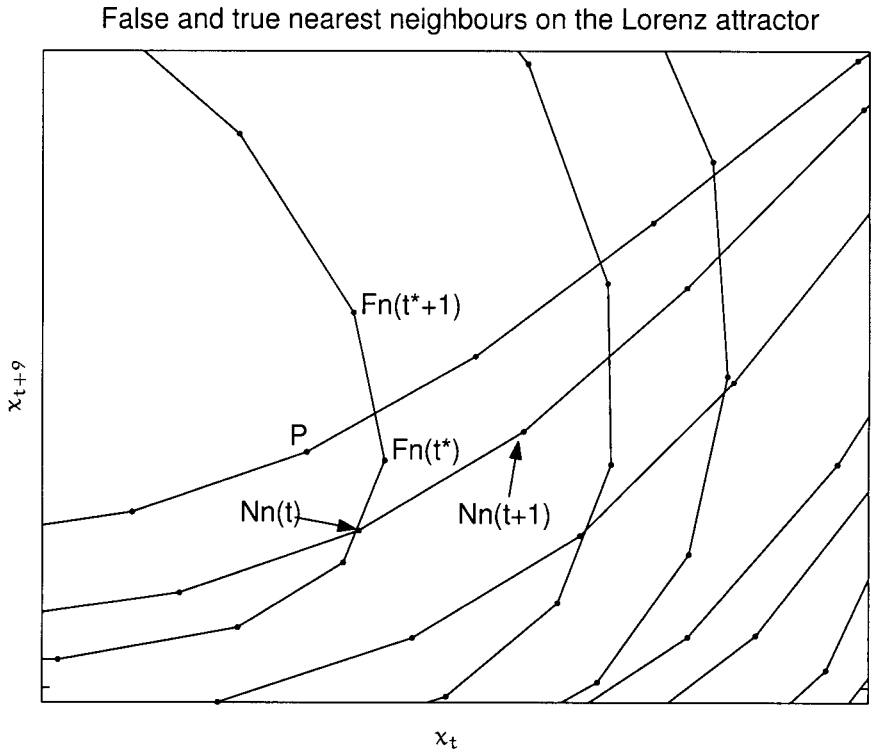

Figure 2.6: An illustration of false and true nearest neighbours.

For the point P the nearest neighbour is the false neighbour $\mathrm{Fn}(t^*)$ as can be seen by the divergence of the false neighbour's trajectory. The real neighbour is point $\mathrm{Nn}(t)$ whose trajectory is similar to that of point P.

The square of Euclidean distance between a point $\mathbf{b}_n$ and a nearest neighbour $\mathbf{b}_r$ in d-dimensional space is

$$[R_d(n,r)]^2 = \sum_{k=0}^{d-1} (x_{n-kl} - x_{r-kl})^2,$$

where l is the chosen time delay. If the embedding dimension is increased by one to $d+1$, then

$$[R_{d+1}(n,r)]^2 - [R_d(n,r)]^2 = [x_{n-dl} - x_{r-dl}]^2.$$

According to Kennel [14], a first of two criteria for a point to be defined as a false neighbour is

$$\frac{|x_{n-dl} - x_{r-dl}|}{R_d(n,r)} > R_{tol},$$

where R_{tol} is a chosen tolerance.

A second criterion is related to the radius of the attractor. If the supplied time series is small a problem might arise because as the embedding dimensions increases, so the points in the attractor spread out to fill the phase space. Thus in small data sets nearest neighbours might not be close, i.e. the Euclidean distance

Dimension	1	2	3	4
Percentage of false nearest neighbours	42.6	1.6	0	0

Table 2.1: Percentage of false nearest neighbours at different embedding dimensions for the Lorenz time series.

between neighbouring points is of the same order as that of the 'radius' of the attractor, R_A, where

$$[R_A]^2 = \frac{1}{N}\sum_{n=1}^{N}\left(x_n - \frac{1}{N}\sum_{n=1}^{N} x_n\right)^2.$$

R_A is then the root mean square value of the data about its mean. Therefore, if a neighbouring point meets the criterion in equation 2.4 it is deemed false, where A_{tol} is a chosen tolerance.

$$\frac{R_{d+1}(n,r)}{R_A} > A_{tol}. \tag{2.4}$$

An important consideration needs to be taken into account if the data is oversampled. Then, when examining a nearest neighbour the point could exist on the same trajectory and just be displaced by the time equivalent of the sampling rate. This point is always a true neighbour and might influence the false neighbour statistic.

A simple solution is to downsample but this discards data and is usually unacceptable. Another solution is to discard points that have a time index within a bound from the index of the point under investigation.

A more elegant solution is that of false strands, as introduced by Kennel [13]. The concept is similar to that of the false nearest neighbour approach except that a group of nearest neighbouring points on a trajectory, or strand, are chosen. If in increasing the dimension any point on a strand is false, the entire strand is deemed false. The method proceeds in a similar fashion to that of the false nearest neighbours method.

Experimental results in chapter 5 illustrate how the choice of an embedding dimensions affects the prediction accuracy.

Chapter 3

Prediction with the Sauer algorithm

To make a series of predictions for a time series, a model of the given time series has to be constructed. The method used for prediction then employs a continuation of the model so that future values of the time series may be calculated by means of some form of extrapolation. Therefore, prediction models illustrate how the time series data must be viewed or interpreted and prediction methods processes the data to make the prediction.

Models may be classified as either local or global models. Global models make predictions of future values of the time series possible by using all known points in the time series. Neural networks (chapter 4) are global models as they are trained on all available time series data to make predictions. Local models in turn only utilize a portion of the attractor in a neighbourhood around the data point for which the prediction is being made. The algorithm devised by Sauer [18] and studied in this chapter is an example of a local model.

The simplest models used for prediction of a time series are based on linear techniques. Linear models only predict periodic oscillating, exponential solutions or combinations thereof.

In turn, non-linear prediction techniques attempt to approximate more of the dynamics of a system and in so doing approximate both the linear and non-linear behaviour of the time series. Due to the flexibility of non-linear models overfitting on data can become a real consideration because the model fits equally well on true time series features and on noise in the time series.

3.1 Premise of Sauer's Algorithm

The Sauer algorithm to be discussed in this chapter has its origins in a very simple but poor prediction technique utilizing embedded attractors that will be termed the center of mass (COM) prediction technique.

Note that for this chapter a trajectory refers to an ordered set of sequential points in a localized area of the attractor around the prediction point. It therefore makes sense to refer to different trajectories on one attractor. This is in contrast to the previously defined view of a single trajectory being defined for the entire attractor.

In short, the COM technique finds a number of nearest neighbours around the last point on the attractor, shifts the nearest neighbours one point forward on the respective trajectories and predicts the next point on the attractor as being the center of mass of these neighbours. This technique is very sensitive to prediction errors and multistep predictions quickly fail.

Sauer takes this simple approach and improves on it by introducing the concept of principal component analysis through the use of singular value decomposition (SVD).

Given an embedded time series (embedding dimension m), the neighbouring points to the final, or prediction, point on the attractor are found. The SVD of the matrix formed by the vectors describing the position of neighbouring points relative to the centre of mass of these neighbouring points is obtained.

A set of orthogonal vectors spanning the row space of a matrix is obtained from the SVD of the matrix. By choosing the first s orthogonal vectors, a subspace is found which is the best s-dimensional linear space ($s \leq m$) through the points in a least-squares sense.

Projections of the matrix onto the column space provided by the SVD are calculated. Linear models are built which map the projections to known future time series values of the neighbouring points. A similar mapping is then applied to the prediction point to predict the next value in the time series.

3.2 Reconstruction of the Attractor

The reconstruction starts with choosing a dimension, m, and time delay, l. If the time series has the same form as previously defined in chapter 2 then the embedding matrix, B, appears as follows,

$$B = \begin{bmatrix} x_{(m-1)l+1} & \cdots & x_{l+1} & x_1 \\ x_{(m-1)l+2} & \cdots & x_{l+2} & x_2 \\ \vdots & & & \vdots \\ x_t & \cdots & x_{t-(m-2)l} & x_{t-(m-1)l} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_{(m-1)l+1}^T \\ \mathbf{b}_{(m-1)l+2}^T \\ \vdots \\ \mathbf{b}_t^T \end{bmatrix}. \tag{3.1}$$

It can be noted from equation 3.1 that the time series is embedded in 'reverse' across the rows of the matrix. This is useful in simplifying some interpretation aspects and does not change the functioning of the algorithm as the net effect is only a linear transformation of the attractor in m dimensions.

The prediction point, $\mathbf{b}_t$, in terms of the embedded time series is the last row, or coordinate point, in B. The target point is $\mathbf{b}_{t+1}$ and is to contain the first predicted value.

Target values, $w_i = x_{(m-1)l+(i+1)}$, are defined as the scalar values such that $\mathbf{b}^T_{(m-1)l+i}$ maps to w_i. It is thus immediately apparent that the prediction point, $\mathbf{b}_t$, maps to the target value $w_t = x_{t+1}$ which is not an embedded value in (3.1).

3.3 Selection of the Nearest Neighbours

The first task is locate the ν nearest neighbours to the embedded prediction point, $\mathbf{b}_t$, from the training set (data points given at the start of the prediction). In extended prediction runs of more than one point it is important to differentiate between the training set and prediction set. The training set is the embedding matrix with which the initial prediction run began. Neighbours are only chosen from the training set so that errors in prediction do not influence the prediction accuracy of later points.

The Euclidean distance from $\mathbf{b}_t$ to any point $\mathbf{b}_i$ in the attractor is calculated as follows

$$\Delta_{ti} = \| \mathbf{b}_t - \mathbf{b}_i \| .$$

The vectors $\mathbf{b}_i$ corresponding to the smallest Δ_{ti}'s are candidates for nearest neighbours. However, only one point on each trajectory should be taken as a nearest neighbour to ensure that enough diverse information can be found about the behaviour of the trajectories in the applicable region of the attractor. Therefore the ν closest neighbours on separate trajectories are chosen as the ν nearest neighbours, and are relabelled, $\mathbf{n}_j$, for $j = 1, \ldots, \nu$. The matrix N is constructed with the ν vectors $\mathbf{n}_j^T$ as rows and their associated target values are relabelled e_j.

$$N = \begin{bmatrix} \mathbf{n}_1^T \\ \mathbf{n}_2^T \\ \vdots \\ \mathbf{n}_\nu^T \end{bmatrix} . \tag{3.2}$$

The target vector is defined as

$$\mathbf{e} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_\nu \end{bmatrix} . \tag{3.3}$$

A method of ensuring a choice of ν neighbours on ν different trajectories is: if any candidate neighbour, $\mathbf{b}_i$ has an index which differs a chosen δ or less from a previously selected neighbour's index then the candidate point is considered to be on a previously selected trajectory and is discarded.

A straightforward way to predict would be to find the best regression of e_j on $\mathbf{n}_j$ and to use it to predict w_t from $\mathbf{b}_t$. However, this method is susceptible to noise in the data. It is therefore more appropriate to draw the more dominant trends from the matrix N and use only that for the regression. The use of the SVD thus becomes prudent and the following sections describe the process involved.

3.4 Center of Mass of the Nearest Neighbours

Having obtained the ν nearest neighbours to $\mathbf{b}_t$ the center of mass of these neighbours has to be found. The normal method for calculating the center of mass vector, $\mathbf{c}$ of a number of points is as follows,

$$\mathbf{c} = \frac{\sum_{j=1}^{\nu} \mathbf{n}_j}{\nu} \qquad i = 1, \ldots, m. \tag{3.4}$$

A weighted centre of mass is used in this thesis, because of slightly improved prediction accuracy, with,

$$\mathbf{c} = \frac{\sum_{j=1}^{\nu} w_j \mathbf{n}_j}{\sum_{j=1}^{\nu} w_j} \qquad i = 1, \ldots, m, \tag{3.5}$$

where w_j is the weight constructed in such a way that the weighting increases with a decrease in the distance of a nearest neighbour to the prediction point. The chosen weighting expression, w_j, is as follows,

$$w_j = \left(1 - \frac{1}{2}\left(\frac{\| \mathbf{n}_j - \mathbf{b}_t \|}{\| \mathbf{n}_\nu - \mathbf{b}_t \|}\right)^2\right)^3.$$

A matrix C is constructed with ν rows of $\mathbf{c}$ as follows,

$$C = \begin{bmatrix} \mathbf{c}^T \\ \mathbf{c}^T \\ \vdots \\ \mathbf{c}^T \end{bmatrix}. \tag{3.6}$$

3.5 Singular Value Decomposition and Projections

Having obtained the matrix C the singular value decomposition, SVD [24], of the resulting displacement matrix, $D = N - C$ can now be calculated. The displacement matrix, D, describes position vectors for each neighbour relative to the centre of mass. By calculating the SVD for the matrix, vectors are obtained that span the row space of the matrix.

Singular value decomposition allows the factorisation of the $\nu \times m$ matrix, D, as follows,

$$D = U\Sigma V^T,$$

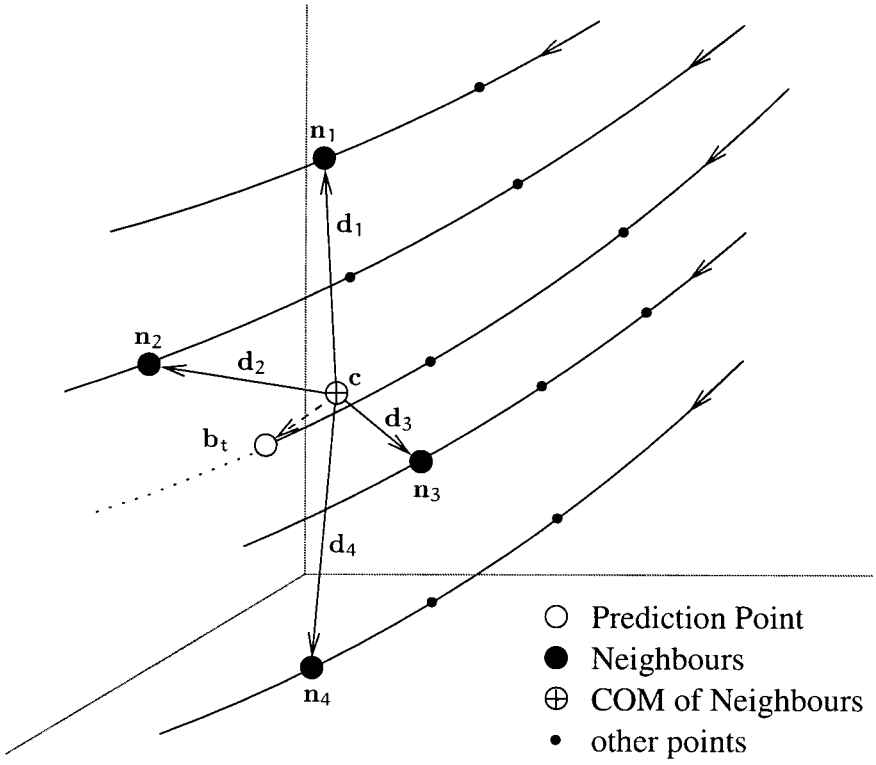

Figure 3.1: Illustration of points on the attractor relevant to the Sauer algorithm.

---

so that

$$U\Sigma V^T = \begin{bmatrix} & & \\ \mathbf{u}_1 & \cdots & \mathbf{u}_\nu \\ & & \end{bmatrix} \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_m \\ \cdots & \cdots & \cdots \\ & \emptyset & \end{bmatrix} \begin{bmatrix} & & \\ \mathbf{v}_1 & \cdots & \mathbf{v}_m \\ & & \end{bmatrix}^T,$$

where U $(\nu \times \nu)$ and V $(m \times m)$ are both orthogonal matrices and Σ $(\nu \times m)$ is diagonal. The vectors, $\mathbf{v}_1, \ldots, \mathbf{v}_m$, are orthonormal bases for the row space of D and $\mathbf{u}_1, \ldots, \mathbf{u}_\nu$ are likewise orthonormal bases for the column space of D.

Having previously chosen a spanning dimension s, so that $s \leq m$, the first s columns in V are selected. The selected vectors are known as the dominant right singular vectors and span the linear space, R^s. R^s is a subspace in the phase space R^m. The spanning vectors,

$$\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_s, \tag{3.7}$$

are packed into the $(m \times s)$ matrix S as follows,

$$S = \begin{bmatrix} & & \\ \mathbf{v}_1 & \cdots & \mathbf{v}_s \\ & & \end{bmatrix}, \tag{3.8}$$

The spanning vectors, S, thus describe a low-dimensional linear space passing through the center of mass, $\mathbf{c}$, of the neighbours.

The advantage of using the SVD becomes apparent because the dynamic information of the displacement vectors are compressed into the first few dominant vectors whereas the non-dominant vectors mainly store less important information such as noise.

Therefore, by projecting the displacement vectors onto the subspace R^s the vectors are forced onto the subspace passing through the centre of mass whilst decreasing the effect of noise in the data.

The projection, P, is calculated as follows

$$P^T = \Theta D^T,$$

where Θ is the projection matrix.

$$\Theta = S\left(S^T S\right)^{-1} S^T.$$

3.6 Regression of the Linear Model

In the following section it is assumed that the discussion revolves around a attractor for which the row count of the embedded matrix is far greater than the dimension m.

Once the columns of D have been projected down to the subspace, R^s, the linear regression model can be found which best fits the projected vectors.

Let Q be the projection matrix P augmented with a column of ones, viz. $Q = [P\ \mathbf{u}]$ with $\mathbf{u} = [1, 1, \ldots, 1]^T$. By augmenting the projection matrix with ones the regression coefficients associated with that column plays the role of a constant. It is thus required to solve the regression coefficients, $\mathbf{k}$, in the following equation,

$$Q\mathbf{k} = \mathbf{e}. \tag{3.9}$$

Expanding (3.9) gives,

$$\begin{bmatrix} P_{1,1} & P_{1,2} & \cdots & P_{1,m} & 1 \\ P_{2,1} & P_{2,2} & \cdots & P_{2,m} & 1 \\ \vdots & & & & \vdots \\ P_{v,1} & P_{v,2} & \cdots & P_{v,m} & 1 \end{bmatrix} \begin{bmatrix} k_1 \\ k_2 \\ \vdots \\ k_{m+1} \end{bmatrix} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_v \end{bmatrix}. \tag{3.10}$$

3.7 Calculation of the Target Point

Having obtained a linear equation for the prediction point $\mathbf{b}_t$ the target point $\mathbf{b}_{t+1}$ can be calculated.

Let $\mathbf{d} = \mathbf{b}_t - c$, then $\mathbf{p}^T = \Theta \mathbf{d}^T$ and $\mathbf{q}^T = [\mathbf{p}^T\ 1]$.

The solution for $\mathbf{k}$ lets $\mathbf{e} \approx Q\mathbf{k}$ such that $\|Q\mathbf{k} - \mathbf{e}\|$ is as small as possible. Therefore the expression $\mathbf{q}^T\mathbf{k}$ approximates $w_t = \mathbf{b}_{t+1}(1)$.

The complete expression for the predicted embedded point $\mathbf{b}_{t+1}$ is

$$\mathbf{b}_{t+1} = \left[\begin{array}{cccc} \mathbf{q}^T\mathbf{k} & \mathbf{b}_{t-l+1}(1) & \mathbf{b}_{t-l+1}(2) & \ldots & \mathbf{b}_{t-l+1}(m-1) \end{array} \right]. \tag{3.11}$$

Note that the 2 through m-th entries in $\mathbf{b}_{t+1}$ (3.11) are previously embedded points in B.

The target point then becomes the prediction point in the next prediction and the algorithm proceeds in the same way as before. It is important to note that predicted points are never used as neighbours.

Chapter 4

Prediction with Time Delayed Neural Networks

4.1 An Introduction to Neural Networks

A neural network [5] [22] [25] [28] is a system of data storage points, called nodes. These nodes are set up in such a way that the values in some of the nodes determine the values in other nodes. If the value in one node depends on the value in another node then these nodes are said to be *connected*. A weight is associated with each connection that determines to which degree the value in the next node is influenced by the value of the previous node.

A neural network always has a set of input nodes and a set of output nodes. It also has some intermediate nodes called *hidden* nodes. The simplest description of the functioning of a neural network (NN) is that it is a nonlinear operator that maps a vector, dimension n, in the input space onto a vector, dimension m, in the output space, where there are n input nodes and m output nodes.

Whether the NN is used for classification or for computational purposes, the user of the NN should have a test space of input vectors with their associated desired output vectors. The test space, or training set, is then used to train the network in the following way: for each input applied to the network the weights are adjusted until the error between the actual output and the correct output is minimised. Once a satisfactorily low error is obtained the network is considered *trained* and the network can then be used to output values for input vectors which have not previously been presented to the model.

The functioning of a neural network is as follows. An input vector introduced to a network is stored in the input layer, (see figure 4.1). The nodes of the input layer are connected to every node in the next layer, the first hidden layer. If all possible connections are made between the nodes in two adjacent layers then the layers are considered fully connected (in graph theory this configuration is referred to as a complete bipartite

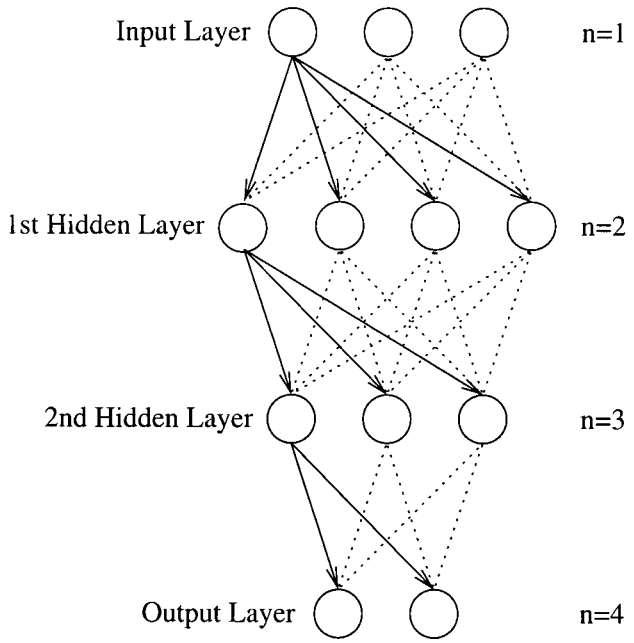

Figure 4.1: Neural network architecture.

graph). This means that the input layer passes all its information to all the nodes in this hidden layer. Each hidden node calculates a weighted sum of all the input values of the input nodes to which it is connected. The weights depend on chosen connection strengths between the two layers. An activation function then scales each resulting sum to within a specified range. There may be more than one hidden layer connected to each other in a similar fashion (Figure 4.1 shows two hidden layers). The hidden layer is similarly fully connected to the output layer and hidden layer values are passed to the output layer where the weighted sums followed by appropriate scaling by activation functions are calculated in the same fashion. This process of calculating node values from input values through to output values will be called *propagation*.

The training process of a NN is briefly as follows. The connection weights are initialised either to small random values or according to a weight initialisation regime (for an example, see [22]). Training input and output data is provided. Input data is propagated through the network. The error between the network's output vector and the correct output vector is calculated and the connection weights are adjusted. This process is repeated for the all input vectors in the training data set. The training set is passed through the network a number of times, each repetition being termed an *epoch*. Once the error is below a certain threshold, the network is considered trained and new data not used during the training process can be introduced.

Neural networks are used for two main types of tasks: classification and mapping. Classification entails the input of a description vector to the network and the output is the subsequent identification. Mapping networks attempt to calculate values for an introduced map variable between different dimensional spaces. Some neural networks are also designed to incorporate both tasks.

Neural networks have the following strengths,

NN's are able to learn temporal dependencies in a system, i.e. systems that responds differently according to different previous states of the system.

NN's are often capable of learning from noisy and incomplete data.

NN's construct approximate solutions to problems for which there is too little information to find a suitable mathematical model.

However, NN's have one notable weakness in that unchecked learning can lead to overfitting of data.

The following characteristics of non-linear time series motivate a neural network approach to prediction,

Time series data obtained from physical processes often contain noise.

A deterministic model describing a non-linear system often cannot be found.

From a theoretical viewpoint, neural networks are well equipped to predict non-linear time-series data [2] [3] [4] [7] [26]. The applicable neural network would function to approximate values of a continuous function with temporal dependant network architecture.

4.2 Structures in Neural Network Architecture

Figure 4.1 illustrates a typical network consisting of an input layer, two hidden layers and an output layer. This network is commonly termed a two-layer network because it has two hidden layers.

The input layer consists of a number of input nodes corresponding to the dimension of the input space. Each input node is connected to some or all of the nodes in the first adjacent hidden layer by connections, each with an associated weight. The subsequent hidden layers are also fully or partially connected to the following layers.

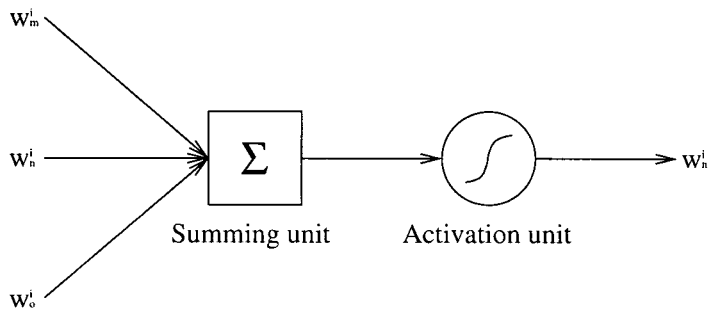

Figure 4.2: The structure of a single neuron.

Consider a multi-layer network with Λ layers (an (Λ−2)-layer network), where L_i^n is the value in the ith node of the nth layer. Let w_{ij}^n be the weight connecting the j-th node in layer $n-1$ with the i-th node in layer n.

The hidden layer and output layer nodes, figure 4.2, are functionally identical, consisting of a summing sub-unit and an activation function sub-unit. The summing sub-unit, for node i in layer n, calculates the following weighted sum,

$$h_i^n = \sum_j w_{ij}^n L_j^{n-1}, \tag{4.1}$$

Activation Functions

The function of the activation sub-unit is to scale the output of each summing sub-unit to within some desired range. The activation sub-unit uses a nonlinear scaling function called the activation function. Apart from its scaling purpose, it also functions to introduce nonlinearity into the network.

General properties of suitable activation functions are as follows

Activation functions are monotonically increasing.

The function domain (input) is real.

The function range (output) is a fixed interval, such as $[0, 1]$ or $[-1, 1]$.

Once the output from the summing sub-unit is passed through the activation function, the hidden node value is as follows,

$$L_i^n = f(h_i^n), \tag{4.2}$$

with h_i^n given in (4.1).

The scaling performed by the activation functions for the output layer are dependant on the scaling regime employed in the training set, i.e. if the training set is scaled to the interval $[0, 1]$ then the output layer activation functions should also be scaled to the same scaling interval. Hidden layer nodes can however have different activation functions and scaling intervals to that of the output nodes. In figure 4.3 the graphs of three well-known activation functions are shown.

The logistic function,

$$f(x) = \frac{1}{1 + e^{-\gamma x}}, \tag{4.3}$$

maps to the interval $[0, 1]$ and is often used for binary output.

For outputs which must be scaled between -1 and 1 a hyperbolic tan,

$$\begin{aligned} f(x) &= \tanh(\gamma x), \\ &= \frac{e^{\gamma x} - e^{-\gamma x}}{e^{\gamma x} + e^{-\gamma x}}, \end{aligned} \tag{4.4}$$

may be used.

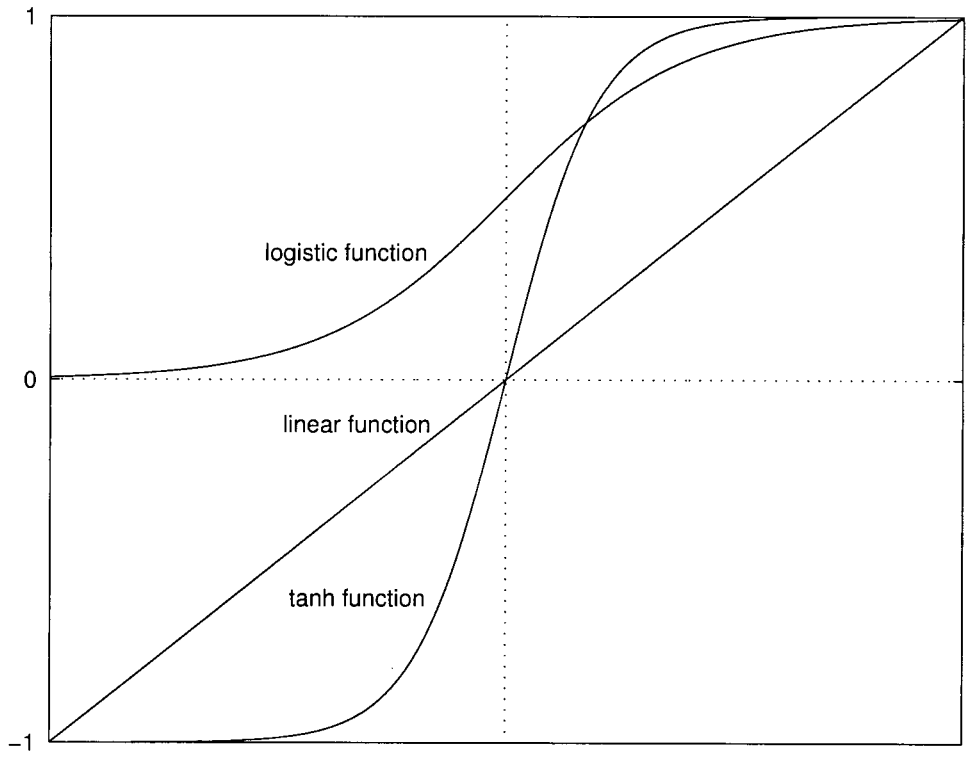

Figure 4.3: Popular activation functions.

Alternatively the linear function,

$$f(x) = \gamma x, \tag{4.5}$$

may be used. It should be noted that the logistic function output is equivalent to the hyperbolic tan function output scaled over the interval $[0, 1]$.

In equations (4.3), (4.4) and (4.5) γ controls the slope of the function.

It would on first sight be tempting to use a linear function of the form

$$f(x) = \gamma x + \frac{1}{2}, \tag{4.6}$$

for binary functions, but this could prove problematic as it is difficult to establish the limits of x (i.e. h_i^n), in the normal functioning of the network. This means that the activation function could give a value of less than 0 or greater than 1 if the value for h_i^n exceeds a certain limit.

The derivatives of the activation functions are used in the process of weight adaptation. The derivative of the logistic function is

$$f'(x) = \frac{\gamma e^{-\gamma x}}{(1 + e^{-\gamma x})^2}, \tag{4.7}$$

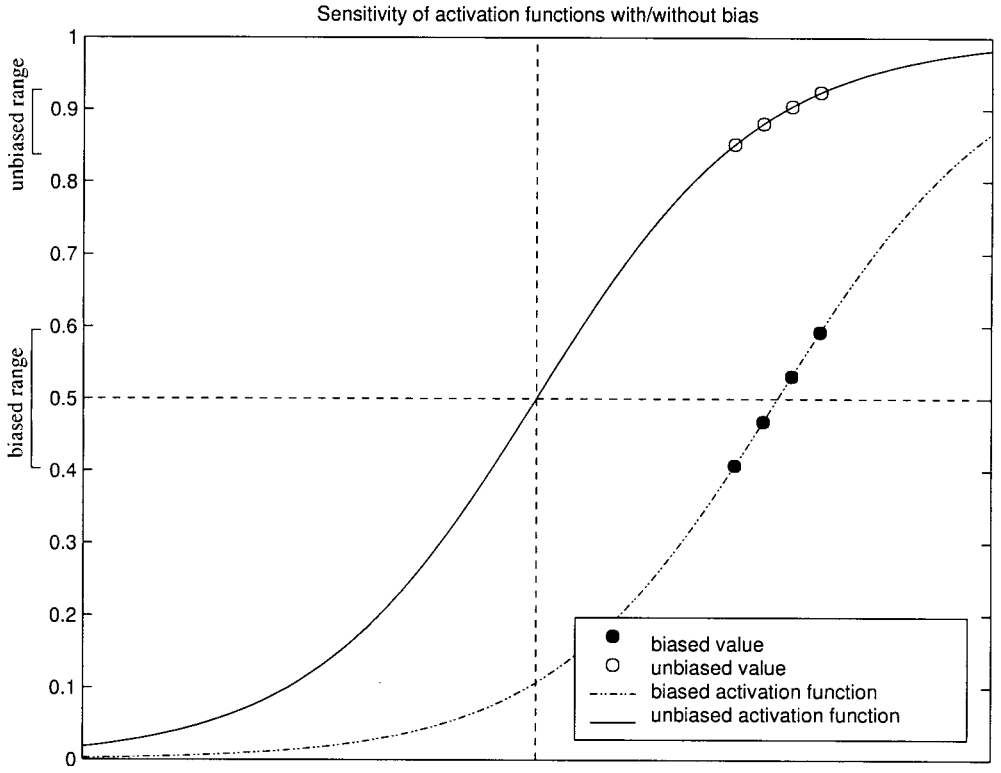

Figure 4.4: The effect of the bias on activation function output.

which can be expressed in terms of the function itself as

$$f'(x) = \gamma f(x)(1 - f(x)). \tag{4.8}$$

The derivative of the hyperbolic tan function can similarly be expressed in terms of the function output, as follows

$$f'(x) = \gamma(1 - f(x)^2). \tag{4.9}$$

Since derivatives of activation functions are needed during training by back-propagation, the choice of the logistic or hyperbolic tan functions are further reinforced by the computationally inexpensive form of their derivatives.

Bias Nodes

For summing sub-unit values that are not close to zero the values obtained from the activation function sub-unit can have a narrow output interval close to the minimum or maximum values of the output interval for the specific activation function, as illustrated by the solid curve in figure 4.4.

A method is thus sought to pull the summing sub-unit values into the range around zero so that the activation function output is over a larger interval closer to the middle of the activation function's output range.

In other words a technique is needed to shift or bias the activation function in the direction of the majority of summing unit values.

Using the logistic activation function as an example, the biased activation function is shifted to the right (figure 4.4), and has the following form,

$$f(x) = \frac{1}{1+e^{-x+b}} \qquad \text{with } b > 0. \tag{4.10}$$

Therefore, for a summing unit value, h, defined in equation (4.1), the activation function has the form,

$$f(h) = \frac{1}{1+e^{-h+b}}. \tag{4.11}$$

If the summing sub-unit value is $\tilde{h} = h - b$, then

$$f(\tilde{h}) = \frac{1}{1+e^{-\tilde{h}}}, \tag{4.12}$$

and

$$\tilde{h}_i^n = w_{i1}^n L_1^{n-1} + w_{i2}^n L_2^{n-1} + \ldots + w_{ij}^n L_j^{n-1} + b. \tag{4.13}$$

Nothing prohibits the decision to consider b in (4.13) as another weight, $w_{i,j+1}^n$. This weight is associated with a node containing an arbitrarily fixed value, c.

Equation (4.13) can then be written as

$$\tilde{h}_i^n = w_{i1}^n L_1^{n-1} + w_{i2}^n L_2^{n-1} + \ldots + w_{ij}^n L_j^{n-1} + w_{i,j+1}^n L_{j+1}^{n-1}, \tag{4.14}$$

where $L_{j+1}^{n-1} = c$. The fixed value of c is usually chosen to be $+1$ or -1.

During training the bias weight is adjusted in the same way as any other weight in order to minimise the network error.

The bias node and its associated weight thus function as an adaptable shifting factor in the activation function to pull values of the summing sub-unit into more sensitive regions of the activation function. It is important to remember that if the nth layer, $(n \leq \Lambda - 1)$, is M^n-dimensional then there is a total of $M^n + 1$ nodes in the layer, the additional node being the bias node.

4.3 Time Dependent Neural Networks

For NN's used for classification purposes the input data is unrelated, i.e. there is not specific order of presentation associated with the data. These types of networks are called *static*. NN's may also be employed to approximate or classify data forming part of a time series. Time series data has a logical order that must be preserved when presented to the NN. This ordered structure motivates the construction of a NN that is able to 'remember' its own earlier states and to utilise that data to influence later states of the network. These NN's are called *Time Dependent Neural Networks*.

A time dependent NN has a complete set of layers for the current time step, just like a static NN. These layers are referred to as the *current layers*. Figure 4.5 shows a time dependent NN with four current layers; and input layer, two hidden layers and an output layer.

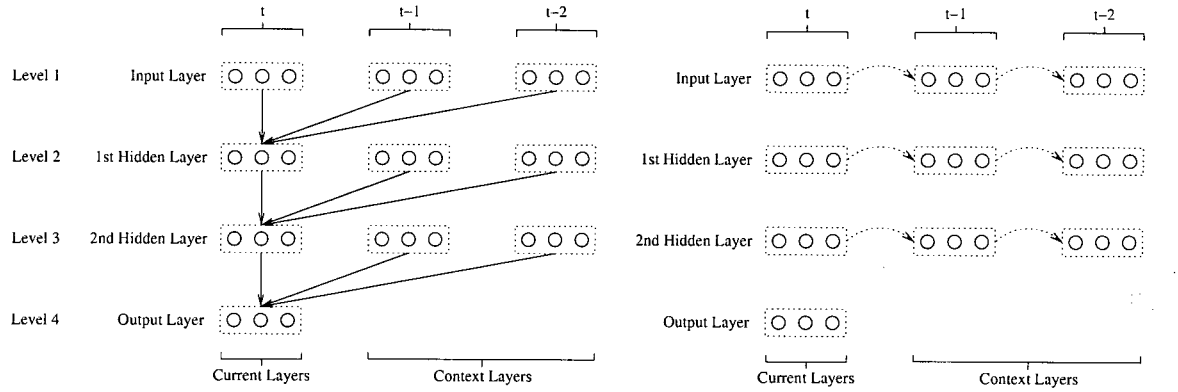

Figure 4.5: **Left:** Feedforward connections for a TDNN. **Right:** Feedback connections for a TDNN.

In order to remember some previous states, a time dependent NN also has some additional sets of nodes, called *taps*. Taps are used to store values of nodes from previous time steps. The values obtained from the nodes being copied to the taps are not altered during the process and in this sense taps can be considered simply as storage registers. A layer of associated tap nodes is referred to as a *context* layer.

A network *level* is defined as the current layer (input, hidden or output) and the context layers associated with it and is named after the current layer's nomenclature. A time dependant NN may have more than one context layer for each associated level. Layers also have time indices, the current layer having time index t and the n-th context layer having the time index $t-n$.

If a current input or hidden layer is $(j+1)$-dimensional with the $(j+1)$-th node being the bias node, then the context layer has j nodes to store the first j node values of the current layer. For output or context layers all nodes in the relevant layers are tapped and the context layer has the same dimension as that of the layer from which it was tapped.

Connections in neural networks can be classified in two groups; *feedback* and *feedforward* connections.

Feedback connections describe the movement of data during the tap updating phase of the training. The context layer nodes contain values that are an element-wise copy of the layer from which they are being copied. The connections between the context layer and the copied layer's nodes thus have to be one-to-one.

Feedforward connections describe the movement of data during the summing and activation phase of training and the connections between layers are usually fully connected. Figure 4.6 illustrates the two connection types.

Feedforward and feedback connections also have specific directional properties. Connections which exhibit

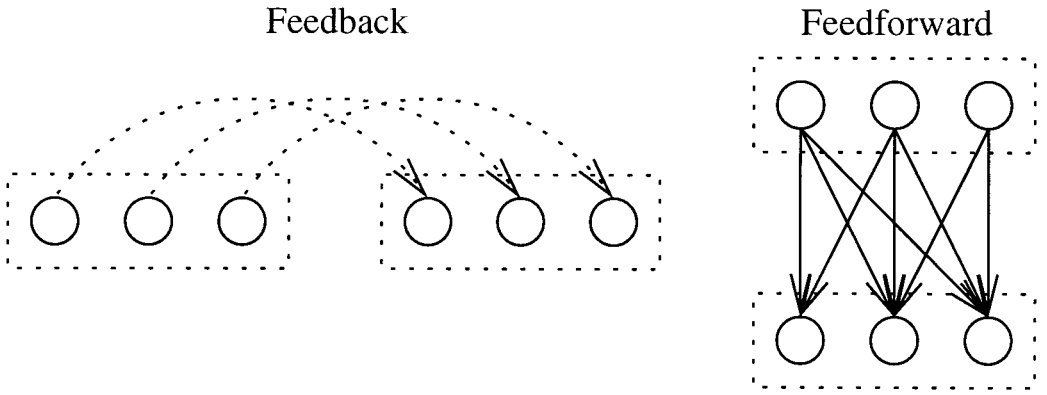

Figure 4.6: Feedback and Feedforward connection detail.

one the of following directional properties are considered feedforward.

Connections are between adjacent current layers in the direction of forward propagation.

Connections are from the first context layer to the current layer in the same level.

Connections are from any context layer to any current layer as long as the current layer's level index is greater than the associated context layer's level index.

It is important to remember that weight adaptation only occurs in reverse along feedforward connections.

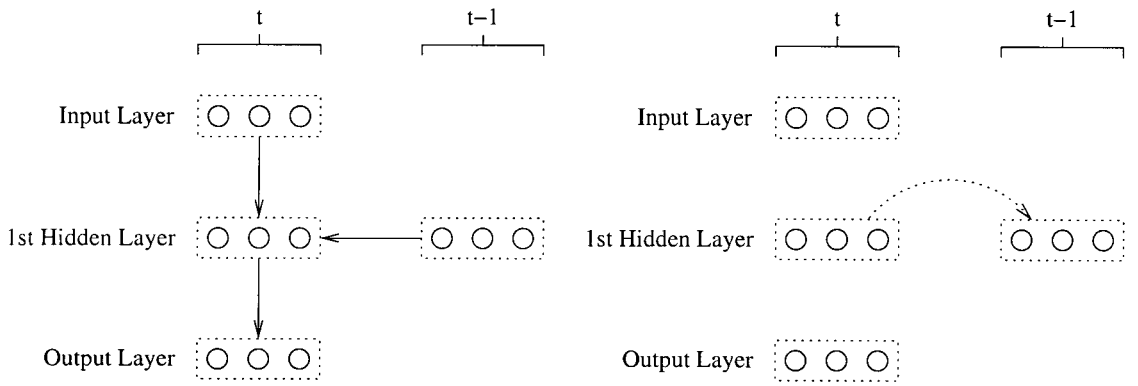

Figure 4.7: **Left:** Feedforward connections for RNN. **Right:** Feedback connections for RNN.

Feedback connections are level specific, in that they describe the movement of node values between the current layer and subsequent context layers. Feedback connections can have the one of the following properties.

Connections are from the current layer to the first context layer in the same level.

Connections are between adjacent context layer's in the same level in the direction of context layer with the earlier time interval.

There are two main types of temporally dependent neural networks; recurrent neural networks (RNN) and time-delay neural networks (TDNN). A recurrent neural network [6] is illustrated in figure 4.7. The single context layer in an RNN is fed back from the same layer to which it is fed forward. Time delay neural networks have context layers which feed forward to layers with level indexes greater than the layer levels from which they are fed back. In figure 4.5 the connectivity between successive layers of a TDNN is illustrated.

4.4 Training Algorithm for Time-Delay Neural Networks

The back-propagation algorithm functions to adapt the set of weights in a fully connected feed forward network so that an output derived from an input introduced to the network is within a certain acceptable error-bound. Gradient descent techniques are used in the adaptation of the weights.

A number of network parameters have to be defined first;

J The number of sets of input patterns, $\Psi(p)$, and associated output patterns, $\Upsilon(p)$, with $p = 1, \ldots, J$.

M^n Dimension of current layer in level n for $n = 1, \ldots, \Lambda$.

H^n Number of context layers in level n, where $n = 1, \ldots, \Lambda$. H^Λ is always set to a value of 0.

The network is first initialised by setting all the weights to small random values and the tap nodes are set to zero.

The pth input pattern, $\Psi(p)$, is applied to the input layer,

$$L_k^1(p) = \Psi_k(p) \qquad k = 1, \ldots, M^1. \tag{4.15}$$

The input is propagated through the network so that

$$L_j^{n+1}(p) = f\left(h_j^{n+1}\right) \qquad n = 1, \ldots, \Lambda - 1, \tag{4.16}$$

where

$$h_i^{n+1} = \left(\sum_{j=1}^{(H^n+1)M^n+1} w_{ij}^{n+1}(p) L_j^n(p) \right) \qquad \begin{cases} i = 1, \ldots, M^{n+1} \\ n = 1, \ldots, \Lambda - 1 \end{cases} \tag{4.17}$$

until the output layer, $L^\Lambda(p)$ has been calculated. In equation (4.17) it should be noted that only the first M^n nodes, in layer i, are connected to all the nodes in the previous layer, j.

Once the input signal has passed through all the nodes in the network the error back-propagation can begin. The proportion of the error used in the updating of the weights for the output layer, δ^Λ, is calculated first by comparing the acquired output, L^Λ, with the expected output, Υ^p.

$$\delta_i^\Lambda = f'\left(L_i^\Lambda(p)\right)\left(\Upsilon_i(p) - L_i^\Lambda(p)\right) \qquad i = 1, \ldots, M^\Lambda, \tag{4.18}$$

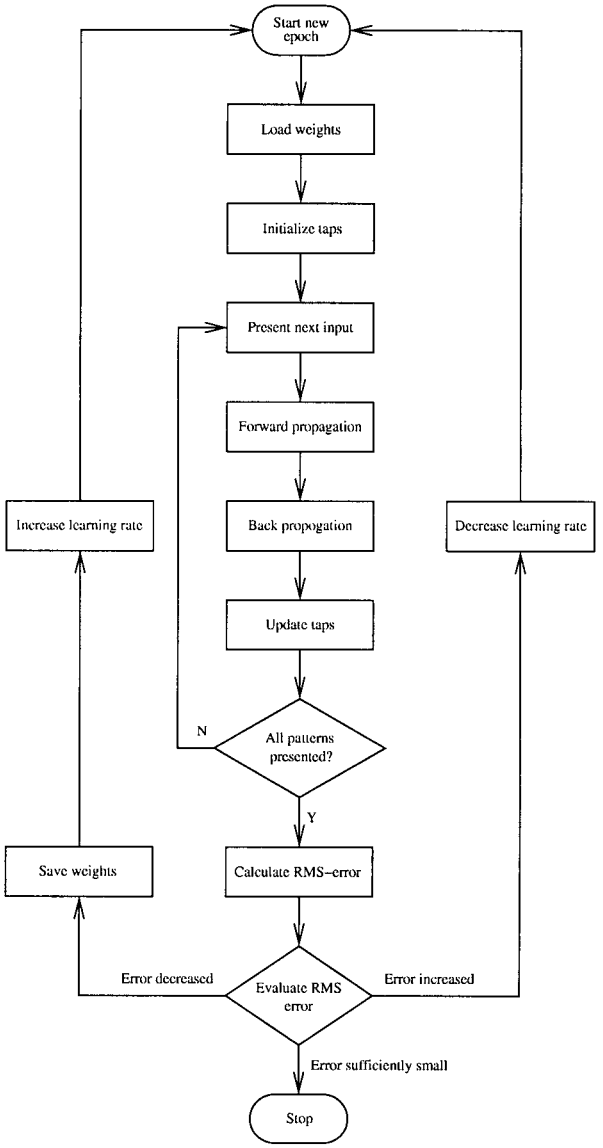

Figure 4.8: Decision tree for adapted back-propagation neural network training.

The output error is then propagated backwards to calculate the errors for the remainder of the network.

$$\delta_i^n = f'(L_i^n(p)) \sum_j w_{ji}^{n+1}(p)\delta_j^{n+1} \qquad n = \Lambda - 1, \ldots, 1, \tag{4.19}$$

Once all the errors have been obtained the weights are updated according to the following formula,

$$w_{ij}^n(p+1) = w_{ij}^n(p) + \alpha \Delta w_{ij}^n(p) + \eta \delta_i^n L_j^{n-1}(\mu) \qquad n = \Lambda - 1, \ldots, 1, \tag{4.20}$$

where

$$\Delta w_{ij}^n(p) = w_{ij}^n(p) - w_{ij}^n(p-1) \qquad n = \Lambda - 1, \ldots, 1, \tag{4.21}$$

In equation (4.20) the learning rate, η, and momentum parameter, α are introduced and will be discussed later in this section.

The back propagation used to obtain experimental results utilizes an adaptive learning rate. If the error obtained with the new weights exceeds the previous error by a certain ratio then the new weights are discarded and the learning rate is decreased. The same input pattern is fed through the network until the error is smaller. If the first test of the error is smaller then the new weights are kept and the learning rate is increased.

Having updated the weights the final task is to update the taps for each layer.

$$L^n_{jM^n+i}(p+1) = L^n_{(j-1)M^n+i}(p) \qquad \begin{cases} i = 1,\ldots,M^n \\ j = H^n - 1,\ldots,1 \end{cases} \tag{4.22}$$

The next pattern can then be introduced to the network and the algorithm is repeated until a satisfactorily small output error is obtained, where typically an RMS-error can be used.

$$(\text{Error}) = \frac{1}{J}\frac{1}{M^\wedge}\sqrt{\sum_{p=1}^{J}\left(\Upsilon(p) - L^\wedge(p)\right)^2} \tag{4.23}$$

The normal back-propagation NN algorithm without taps differs from the algorithm used here in two ways; t^n in (4.17) has a value of 0 and (4.22) is omitted.

Gradient Descent and Learning Parameters

Back-propagation functions to adjust the weights, w^n_{ij}, by performing a gradient descent along the error surface pertaining to the specific position of the weight in the network. Thus each weight, w^n_{ij}, is adjusted by an amount, Δw^n_{ij}, which is proportional to the gradient of the error function at the specific point on the error surface,

$$\Delta w^n_{ij} = -\eta\frac{\partial E}{\partial w^n_{ij}}. \tag{4.24}$$

where η is the learning rate used to dictate the proportion of the error that is used to update the weight. The error function for a multi-layer network is as follows

$$\begin{aligned} E(\underline{w}) &= \sum_{\mu=1}^{J}\left(\Upsilon(\mu) - L^\wedge(\mu)\right)^2 \\ &= \sum_{\mu=1}^{J}\left(\Upsilon(\mu) - f\left(\sum_{j=1}^{(H^{\wedge-1}+1)M^{\wedge-1}+1} w^\wedge_{ij}L^{\wedge-1}_j(\mu)\right)\right) \\ &= \sum_{\mu=1}^{J}\left(\Upsilon(\mu) - f\left(\sum_{j=1}^{(H^{\wedge-1}+1)M^{\wedge-1}+1} w^\wedge_{ij} f\left(\ldots f\left(\sum_{j=1}^{(H^0+1)M^0+1} w^1_{ij}L^0_j(\mu)\right)\right)\right)\right). \end{aligned} \tag{4.25}$$

Equation (4.25) illustrates the way in which weights are embedded in the error function. For the weights connecting the hidden and output layers equation (4.24) has the following form

$$\Delta w^\wedge_{ij} = -\eta\frac{\partial E}{\partial w^\wedge_{ij}}$$

$$\begin{aligned} &= \eta \left[f'\left(h_i^{\wedge}\right)\left(\Upsilon_i(\mu) - L_i^{\wedge}(\mu)\right)\right] \cdot L_j^{\wedge-1}(\mu) \\ &= \eta \delta_i^{\wedge} \cdot L_j^{\wedge-1}(\mu). \end{aligned} \tag{4.26}$$

Weights in lower layers of the network are nested more deeply in equation (4.25). The differentiation thus proceeds differently, so that by the chain rule

$$\begin{aligned} \underline{\Delta} w_{jk}^{\wedge-1} &= -\eta \frac{\partial E}{\partial w_{jk}^{n}} \\ &= -\eta \sum_{\mu} \frac{\partial E}{\partial L_j^{\wedge}(\mu)} \frac{\partial L_j^{\wedge}(\mu)}{\partial w_{jk}^{n}} \\ &= \eta \sum_{\mu i} f'\left(h_i^{\wedge}\right)\left(\Upsilon_i(\mu) - L_i^{\wedge}(\mu)\right) w_{ij}^{\wedge} f'(h_j^{\wedge-1}) L_k^{\wedge-1}(\mu) \\ &= \eta \sum_{\mu i} \delta_i^{\wedge} w_{ij}^{\wedge} f'(h_j^{\wedge-1}) L_k^{\wedge-1}(\mu) \\ &= \eta \sum_{\mu} f'(h_j^{\wedge-1}) \left(\sum_i w_{ij}^{\wedge} \delta_i^{\wedge} \right) L_k^{\wedge-1}(\mu) \\ &= \eta \sum_{\mu} \delta_j^{\wedge-1} L_k^{\wedge-1}(\mu) \end{aligned} \tag{4.27}$$

Having obtained the amount, $\underline{\Delta} w_{ij}^{n}$, by which each weight has to be updated, the updating rule can be set out

$$\begin{aligned} w_{ij}^{n}(\mu+1) &= w_{ij}^{n}(\mu) + \underline{\Delta} w_{ij}^{n}(\mu) \\ &= w_{ij}^{n}(\mu) + \eta \delta_i^{n} L_j^{n-1}(\mu) \end{aligned} \tag{4.28}$$

The learning rate, η, and momentum, α, determine the speed and flexibility of the gradient descent. The learning rate functions to scale the error obtained for each update. This functions to balance stability against learning speed. A high value for η will improve learning speed but may lead to overshooting of the solution, and hence oscillation around the solution point. A low value for η will avoid overstepping but may result in the gradient descent becoming trapped in a local minimum on the error slope.

If the error surface were smooth, it would be a simple task to train the network to obtain a minimum output error. The error surface is however usually not smooth and usually contains local minima which may cause the values of the weights to become trapped for certain values of η. A method is required to step through these local minima. This is accomplished by adding a momentum term to the weight update equation, where the momentum is a proportion, α, of the difference between the previous two weight values.

The complete weight adjustment equation is then as follows,

$$w_{ij}^{n}(\mu+1) = w_{ij}^{n}(\mu) + \alpha \Delta w_{ij}^{n}(\mu) + \eta \delta_i^{n} L_j^{n-1}(\mu) \tag{4.29}$$

where

$$\Delta w_{ij}^{n}(\mu) = w_{ij}^{n}(\mu) - w_{ij}^{n}(\mu-1). \tag{4.30}$$

Chapter 5

Evaluation of the algorithms

The question posed in this thesis is: given a portion of a time series that is known to exhibit deterministic chaotic behaviour, how accurately can future values and behaviour of the time series be predicted by the two chosen algorithms?

For the purposes of testing an algorithm the supplied time series is divided into two parts, the larger section of the time series is used for training the model. The second, smaller portion of the time series is used to compare to the predicted time series to assess the accuracy of the prediction and is termed the test set.

Both the Sauer and TDNN algorithms utilize a reconstructed attractor and therefore can be tested for the degree of prediction performance as it relates to the choice of embedding parameters. Further algorithm specific parameters effects on prediction performance can also be tested, such as the value for the spanning dimension and number of neighbouring points in the Sauer algorithm and the number of input and hidden taps in the TDNN algorithm.

Predictions were free-run in that the predicted values were reintroduced to the algorithm as the next prediction point. This is far more demanding on the performance of the algorithms in that prediction errors influence future prediction steps. Figure 5.1 illustrates a free-run prediction and the effect of the compounding error.

As has been mentioned before, the performances of the algorithms are evaluated under two criteria. The first criteria relates to the degree to which the algorithm captures the dynamics of the time series, i.e. if the Lorenz time series is being predicted does the predicted time series resemble the typical behaviour of the Lorenz time series? This criterion is independent of the second criterion, that of the prediction accuracy.

It might initially be felt that information obtained from predictions utilizing a reconstructed attractor may only be meaningful and representative if the reconstruction uses the correct embedding dimension and time delay. However, it will be shown that acceptably accurate predictions, with non-optimal values of the embedding parameters, can still be made.

Chapter 2 highlighted the methodologies behind obtaining the minimal embedding dimensions from the false

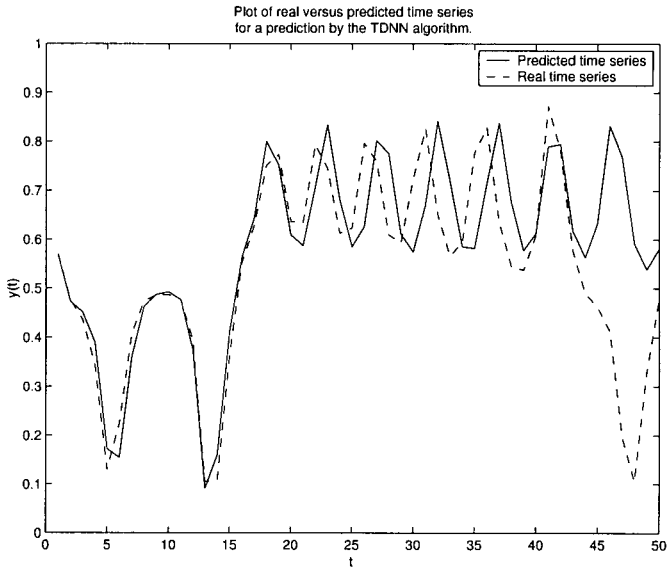

Figure 5.1: Plot illustrating a predicted time series versus the associated real time series obtained from a prediction by the TDNN algorithm.

---

nearest neighbours and the optimal time delay from the average mutual information of a time series. For the Lorenz time series the minimal embedding dimension is 3 and the optimal time delay is 9, as presented in figure 2.5 and table 2.1 respectively.

5.1 Evaluation of the TDNN algorithm

Both the time-delay neural network and Sauer algorithms are presented with the embedded data from the first 1200 points of a scaled (see Appendix A) Lorenz time series.

The experimental parameters for the TDNN algorithm appear in table 5.1. All possible combinations of m,

Parameter	Testing values
Embedding Dimension (m)	[1]
Time delay (l)	$[6, \ldots, 11]$
Number of hidden layer units (H)	[120]
Number of input taps (t_I)	$[1, \ldots, 5]$
Number of hidden taps (t_H)	[0, 1]

Table 5.1: Testing parameters for TDNN algorithm.

l, H, t_I and t_H were tested for the TDNN algorithm resulting in the evaluation of 60 different parameter combinations.

Given a time series,

$$T = x_1, x_2, x_3, \ldots, x_{n-1}, x_n, \tag{5.1}$$

the input data for the TDNN has a different format to that of the normally embedded time series (see equation 3.1). For a specific time-delay l the time series presented to the TDNN is as follows,

$$T = x_1, x_{1+l}, x_{1+2l}, \ldots, x_{1+(n-1)l}, x_{1+nl}. \tag{5.2}$$

The TDNN is thus presented with one entry from (5.2) at a time, which is tapped by the TDNN to effectively form a higher dimensional embedding space.

5.1.1 Performance Criteria

The success of a prediction attempt is assessed by the following two criteria:

1. Did the algorithm capture the dynamics of the time series being predicted?
2. How accurate was the prediction of the time series?

Criterion 1: Qualitative behaviour.

The first criterion is used to assess whether the time series created by the prediction has the same characteristics as the time series being predicted. A predicted time series is said to have captured the dynamics of the training time series if the predicted time series is qualitatively similar to the training time series in some sense.

Methods were sought to obtain characteristic features from the solution of the Lorenz equations.

The first method obtained was based on the fact that the Lorenz attractor has an easily recognizable form in embedding space, i.e. the butterfly plot. A point following the trajectory on the Lorenz attractor will orbit either on the one 'wing' or on the second 'wing' by switching between the two wings through the transition. It should be noted that a transition is defined when a point on the trajectory leaves one wing to orbit the second wing at least once.

The characteristic qualities in the Lorenz time series are the number of peaks above the transitions (c_p), number of troughs below the transitions (c_v) and the number of transitions (c_t). A simple search algorithm is used to find the number of turning points n_t in the time series, i.e. all local minima and maxima. The vector

$$\left[\frac{c_p}{n_t}\ \frac{c_v}{n_t}\ \frac{c_t}{n_t}\right], \tag{5.3}$$

describes a point in characteristic space, which will be termed C-space henceforth.

All Lorenz time series of a fixed length will not have the same point in this three dimensional C-space because

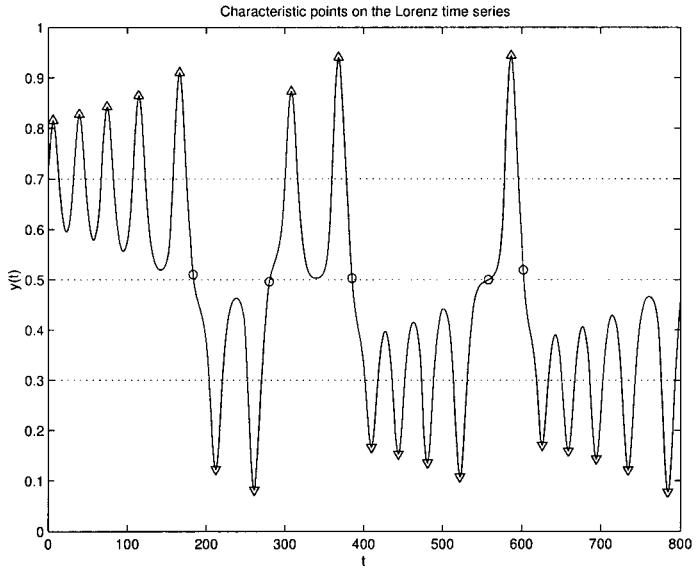

Figure 5.2: Characteristic points for the Lorenz time series. [$\triangle$: Peaks above the transition, ∇ : peaks below the transition, $\circ$: transitions]

---

of the chaotic nature of the time series. These points do however occupy a localized cloud in C-space as is illustrated in figure 5.4.

Therefore, any predicted time series will also possess some or all of these characteristic qualities and possess a point in C-space. An Euclidean distance measure is used to calculate the distance from the C-space point of the prediction to the closest C-space point of the training set. A predicted time series that captures the dynamics of the Lorenz time series will thus have a point close to the real-point cloud.

The second method employed was a statistical approach based on the relationship between the Lorenz time series and a numerical approximation if its first derivative, see figure 5.5.

The process starts by calculating the backward difference,

$$f_i = \frac{x_i - x_{i-1}}{h} \tag{5.4}$$

for each point x_i. Figure 5.5 shows a plot of f_i versus x_i. The space (f_i, x_i) is then divided into regular bins and the number of points that fall within each bin, ij are tallied. The results are assembled into the bin matrix b_{ij}.

Given two time series R_1 and R_2, how can the "closeness" of the two time series be measured from the relevant bin matrices?

Let K for the two time series be,

$$K = \sum_i \sum_j \left(b_{ij}^{R_1} - b_{ij}^{R_2} \right)^2, \tag{5.5}$$

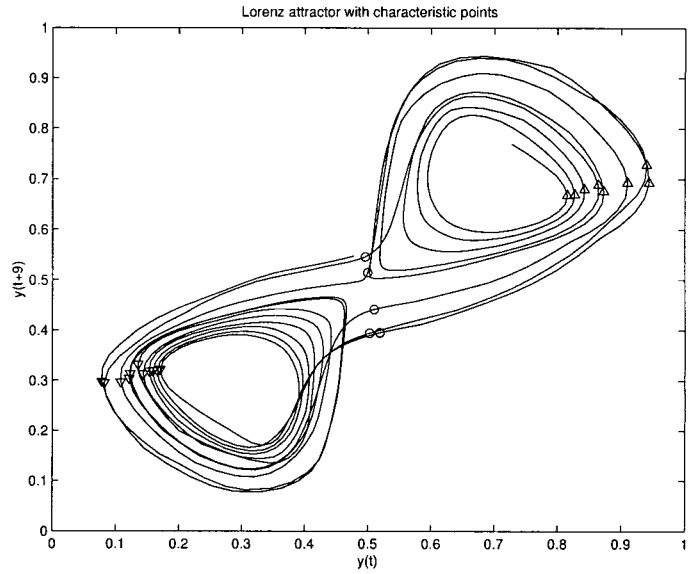

Figure 5.3: Characteristic points on the Lorenz attractor. [△ : Peaks above the transition, ▽ : peaks below the transition, ∘ : transitions]

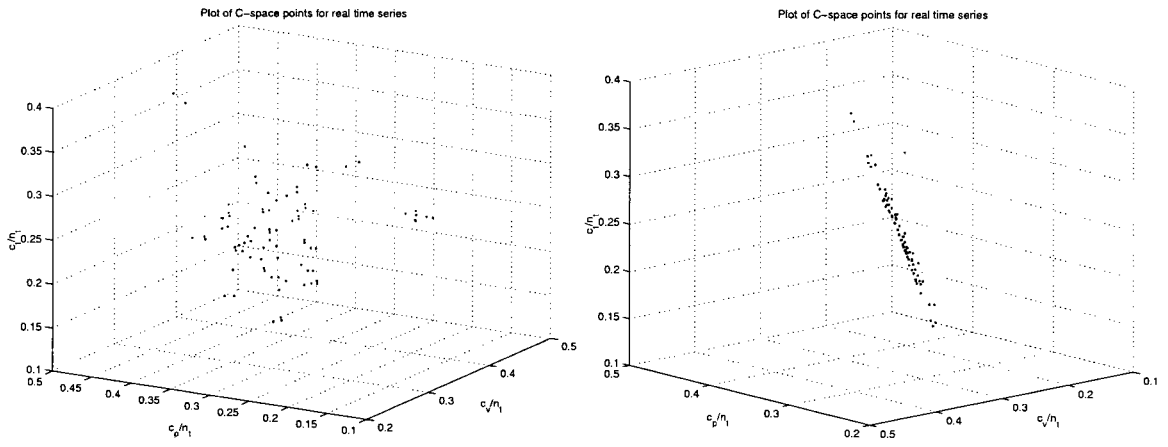

Figure 5.4: Plot of the C-space points for real time series from two viewpoints.

where time series with a value of K that is close to zero are considered to be similar to each other in terms of dynamics.

Given a set of predicted attractors the method proceeds similarly by calculating the set of K values for the combination of a predicted attractor with each real attractor.

A further characteristic of the derivative attractor is required before predicted attractors can be accepted or rejected. From figure 5.5 it can be seen that the attractor is near-symmetrical around the 0.5 on the x-axis and 0 on the y-axis. Therefore, a further characteristic is the ratio of points, ρ_x and ρ_y, either side of the lines of symmetry.

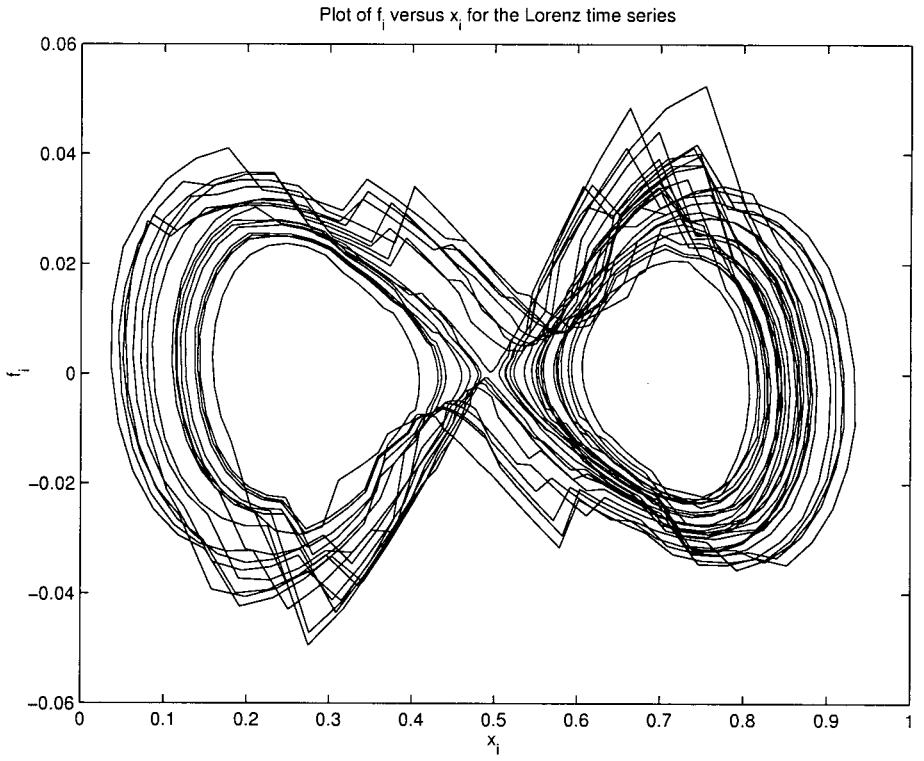

Figure 5.5: Plot of the first derivative of the Lorenz time series versus the Lorenz time series.

The symmetry ratios together with the value for K describe a point, [K ρ_x ρ_y], in three dimensional fx-space. In a similar method as to the C-space method, the Euclidean distances between real and predicted points in the fx-space are calculated.

Criterion 2: Prediction accuracy.

It has to be borne in mind that the behaviour criterion is tested over the entire length of the predicted time series, in this thesis the prediction length was 300 points, whereas the accuracy criterion is tested over a small number of time series values, in this thesis the first 30 points.

At first glance, this might indicate that the behaviour criterion is more stringent than the accuracy criterion but in practice this is desirable because a trained neural network is sought that is significantly insensitive to prediction errors. As an example, a neural network that captures the dynamics of a time series for the first 20 points but then dissipates for further points is not as well trained as a neural network that only becomes affected by prediction errors after 300 points.

The prediction accuracy is measured with the mean square error function [15], $\tilde{s}(E)$, where E is the vector of

element wise errors between the predicted time series and the real time series. Therefore,

$$\tilde{s}(E) = \sqrt{\frac{1}{n}\sum_{j=1}^{n}(E_j - \bar{E})^2}, \tag{5.6}$$

with n being the length of the predicted time series and $\bar{E}$ the mean of the errors.

5.1.2 Experimental results obtained for the TDNN algorithm.

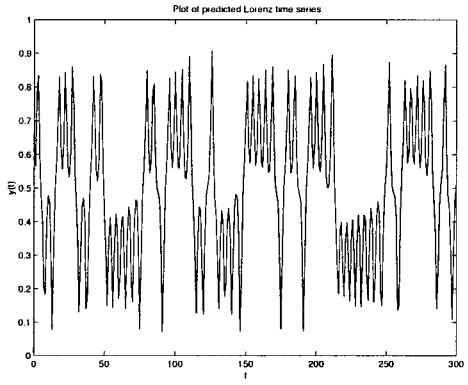

Figure 5.6: Plot of a qualitatively correct simulation.

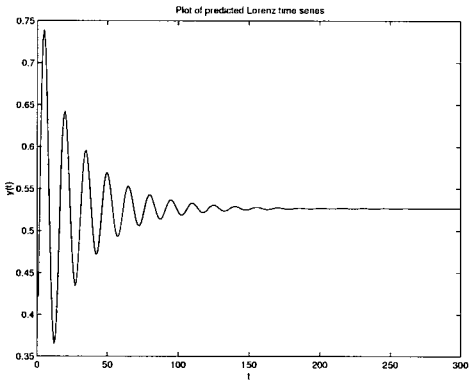

Figure 5.7: Plot of a prediction that decays towards a false fixed point.

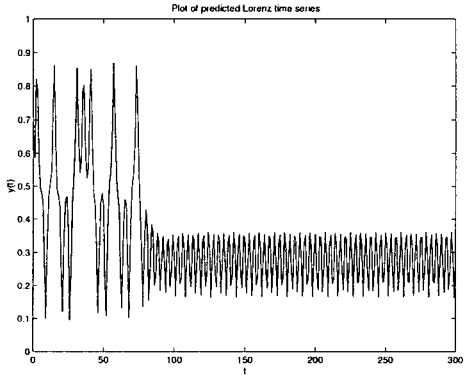

Figure 5.8: Plot of a prediction that first captures the dynamics but is trapped.

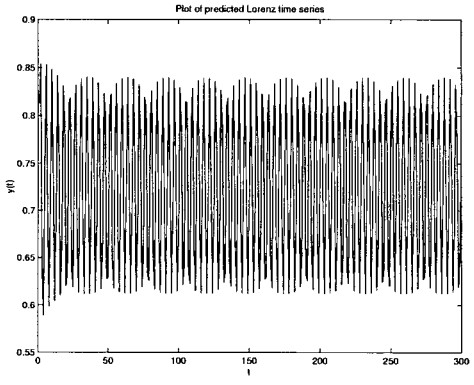

Figure 5.9: Plot of a prediction that displays oscillatory behaviour.

Predictions were performed for all combinations of allowable dimension and delay with TDNN's trained for 500 epochs.

The neural network's simulation of the Lorenz time series can be broken into four main groups; accurate simulation, decay towards a false fixed point, delayed single wing trajectory capture or periodic oscillations. The four classes of emulation appear in figures 5.6, 5.7, 5.8 and 5.9.

5.1.3 Analysis of results for the TDNN algorithm.

Five repetitions of the algorithm were performed for each combination of the parameters. Each repetition had training data of a 1200 values from a different section of the Lorenz time series. For each of these repetitions the TDNN was trained over 500 epochs. After each epoch, a prediction run of 300 values was calculated, and these were evaluated by means of the two criteria.

Out of each set of 500 prediction runs the best performing simulation was selected and only the results of these simulations were used for comparison. It should be noted that the 500th epoch was not necessarily the best performing simulation.

The tables to follow were constructed by finding the 5 instances for each combination of embedding dimension and time delay and calculating the relevant performance measure. The performance measure was obtained by calculating the mean of the five MSE or Euclidean norms values corresponding to the chosen parameters. The means of the performance measures for the rows and columns in the tables were also calculated.

Note that in all tables of results bold numbering indicates the minimum value for each delay and bracketed values indicate the minimum value for each dimension. The underlined values indicate the minimum values for the means of all dimensions and delays.

Dimension	Time delay						
	6	7	8	9	10	11	mean
2	0.19	0.18	0.19	0.19	(0.17)	**0.18**	0.18
3	0.19	(0.16)	0.18	0.19	0.17	**0.18**	0.18
4	0.16	0.17	(**0.15**)	0.18	(**0.15**)	**0.18**	0.17
5	(**0.15**)	0.16	0.16	**0.16**	0.17	**0.18**	<u>0.16</u>
6	**0.15**	(**0.13**)	0.16	0.18	0.17	**0.18**	<u>0.16</u>
mean	0.17	<u>0.16</u>	0.17	0.18	<u>0.16</u>	0.18	

Table 5.2: Average MSE values for TDNN predictions using accuracy criterion.

Dimension	Time delay						mean
	6	7	8	9	10	11	
2	0.32	0.38	0.33	0.33	0.35	(**0.13**)	0.31
3	0.20	(0.15)	(0.15)	0.27	0.41	0.22	0.23
4	0.26	0.29	**0.13**	0.25	(**0.12**)	0.32	0.23
5	(0.16)	0.27	(0.16)	**0.18**	(0.16)	0.18	<u>0.19</u>
6	(**0.10**)	**0.12**	0.17	0.23	0.29	0.36	0.21
mean	0.21	0.24	<u>0.19</u>	0.25	0.26	0.24	

Table 5.3: Average euclidean distance by TDNN prediction for the behaviour criterion using the fx-space discrimination.

Two sets of experiments were done. The first set of experiments was for a TDNN with input context layers and no hidden context layers. The results are presented in tables 5.2, 5.4 and 5.3. The second set was for a TDNN with input context layers and one hidden context layer. The results are presented in tables 5.5, 5.6

Dimension	Time delay						
	6	7	8	9	10	11	mean
2	0.60	1.07	(0.53)	0.75	1.85	1.37	1.03
3	(**0.18**)	0.20	0.23	0.70	2.96	1.33	0.94
4	1.45	1.16	(**0.22**)	0.70	**0.33**	1.76	0.93
5	(0.22)	0.69	0.24	**0.51**	0.83	**0.88**	0.56
6	0.37	(**0.18**)	0.39	1.14	1.67	1.87	0.94
mean	0.56	0.66	0.32	0.76	1.53	1.44	

Table 5.4: Average euclidean distance by TDNN prediction for the behaviour criterion using the C-space discrimination.

Dimension	Time delay						
	6	7	8	9	10	11	mean
2	0.18	(0.15)	0.19	0.18	0.16	0.18	0.17
3	0.18	(0.16)	**0.17**	0.17	0.17	0.18	0.17
4	(**0.15**)	0.17	0.18	0.16	(**0.15**)	0.18	0.16
5	**0.15**	0.15	0.18	0.18	0.16	(**0.14**)	0.16
6	**0.15**	**0.14**	0.18	(**0.13**)	0.17	0.17	0.16
mean	0.16	0.15	0.18	0.17	0.16	0.17	

Table 5.5: Average MSE values for TDNN predictions using accuracy criterion with hidden context layers.

Dimension	Time delay						
	6	7	8	9	10	11	mean
2	0.39	(0.35)	0.40	(0.35)	(0.35)	0.42	0.37
3	0.36	0.31	(**0.24**)	0.33	0.33	0.43	0.33
4	(0.09)	0.33	0.32	0.30	**0.26**	0.42	0.29
5	(0.20)	0.27	0.39	0.22	0.30	**0.26**	0.27
6	**0.08**	**0.14**	0.35	(**0.07**)	**0.26**	0.30	0.20
mean	0.22	0.27	0.34	0.25	0.30	0.37	

Table 5.6: Average euclidean distance by TDNN prediction with hidden context layers for the behaviour criterion using the C-space discrimination.

Dimension	Time delay						mean
	6	7	8	9	10	11	
2	2.70	(1.58)	3.34	2.00	2.67	3.70	2.66
3	1.86	1.60	(**1.47**)	1.63	2.66	3.23	2.08
4	(**0.08**)	0.90	2.75	1.91	2.93	3.41	2.00
5	0.76	(0.31)	2.30	0.53	1.36	**1.26**	1.09
6	(0.16)	**0.25**	1.52	**0.10**	**0.56**	1.49	0.68
mean	1.11	0.93	2.28	1.23	2.04	2.62	

Table 5.7: Average euclidean distance by TDNN prediction with hidden context layers for the behaviour criterion using the fx-space discrimination.

and 5.7.

Experiments without hidden context layers.

The false nearest neighbours method (chapter 2.3) for the obtaining the minimal embedding dimension gave an embedding dimension of 3 as being the minimal sufficient value. From table 5.2 it can be seen that for criterion 2 the embedding dimensions of 2 and 3 have the highest mean for all time delays. The smallest mean MSE values are obtained for dimensions 5 and 6. This is in agreement with the fact that the minimal embedding dimension value is 3 and that higher values of the embedding dimension provide more information about the time series.

Average mutual information (chapter 2.2) for the Lorenz time series indicates an optimal time delay of 9 (figure 2.5). The range over which the values for the average mutual information is distributed for time delay values between 7 and 10 is small enough to warrant accepting time delays in this window. This is borne out by the experimental results where time delays of 7 and 10 provide the most accurate predictions.

Table 5.2 also shows the optimal combinations of embedding dimension and time delay. It can be seen that there is no discernable relationship between dimension and time delay.

Results for the behaviour criterion, in tables 5.3 and 5.4, illustrate that the smallest mean embedding dimension regardless of time delay occurs for a value of 5 for both methods. Similarly, the smallest mean time delay regardless of embedding dimension occurs for a value of 8 for both methods.

Therefore, a dimension of 5 satisfies both criteria but the values for the delay differ between the criteria. It can however be noted that the range over which the mean values for the accuracy criterion are distributed is sufficiently small to argue that a delay of 8 would satisfy the accuracy criterion.

Experiments with hidden context layers.

For the accuracy criterion , in tables 5.2 and 5.5, it can be noted that, for the mean MSE values (last column and last row), the TDNN with a hidden context layer performs better than a TDNN without a hidden context layer.

It should however be noted that the TDNN with hidden context layers performs worse under the behaviour criterion, as can be noted by the mean Euclidean values in comparing tables 5.3 and 5.7 as well as 5.4 and 5.6.

5.2 Evaluation of the Sauer algorithm.

Parameters for the Sauer algorithm given in table 5.8, and for each embedding dimension, m, all possible combinations of l, N and $s \leq m$ were tested.

Parameter	Testing values
Embedding Dimension (m)	[2,...,6]
Time delay (l)	[6,...,11]
Number of nearest neighbours (ν)	[12,...,24]
Spanning dimension (s)	[2,...,m]

Table 5.8: Testing parameters for Sauer algorithm.

Given a time series,

$$T = x_1, x_2, x_3, \ldots, x_{n-1}, x_n. \tag{5.7}$$

The coordinate point matrix of the attractor constructed for the Sauer algorithm appears in (3.1).

5.2.1 Performance Criteria

Having formulated the use of the C-space and fx-space methods for the evaluation of the behaviour criterion in section 5.1.1 it is now necessary to highlight a shortcoming of both methods, in that they rely on time series that is statistically representative. For example, if a supplied portion of the Lorenz time series does not contain the correct ratio of peaks, transition and troughs then the two methods will not correctly identify the validity of the time series.

This limitation comes to the fore when implementing the Sauer algorithm. For the TDNN algorithm the neural network predicted a time-delayed time series of 300 points. If the chosen time-delay was l then the time-delayed time series actually represented a time series of length 300l. The Sauer algorithm in turn predicts a time series of 300 points that is not time delayed. The predicted time series is therefore too short for evaluation by the previous methods.

A new method is sought to evaluate the Sauer predicted time series. A method was employed that calculated the mean squared error values between the predicted time series and a collection of time series sections obtained directly from the Lorenz time series for 300 points. Due to the fact that test sets contained between 8 and 12 pseudo-periods the method proved robust against noise. Had the test sets contained more pseudo-periods, noise could have been misinterpreted as legitimate oscillations of the Lorenz time series. Although this method is brute force it did identify predicted time series that captured the dynamics of the system.

The accuracy criterion proceeds as per the process used for evaluation of the TDNN algorithm.

5.2.2 Experimental results obtained for the Sauer algorithm.

The Sauer algorithm's simulation of the Lorenz time series can be divided into 4 classes; accurate simulation, delayed attraction to a sink, periodic oscillations and divergence from the attractor. These behaviours are

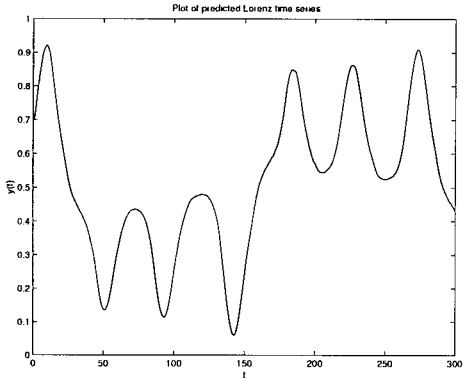

Figure 5.10: Plot of a qualitatively correct simulation by the Sauer algorithm.

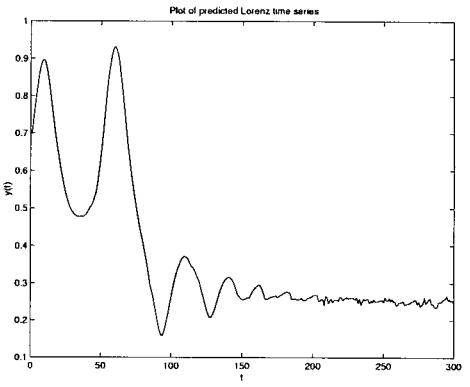

Figure 5.11: Plot of a prediction by the Sauer algorithm that is trapped on one wing.

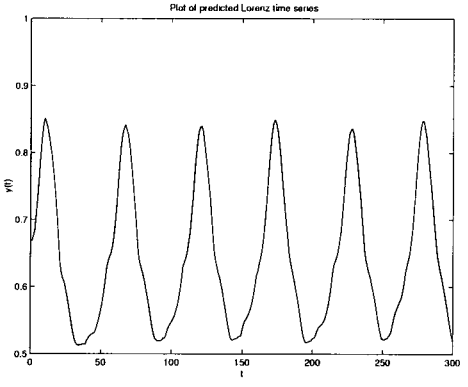

Figure 5.12: Plot of a prediction by the Sauer algorithm that displays oscillatory behaviour.

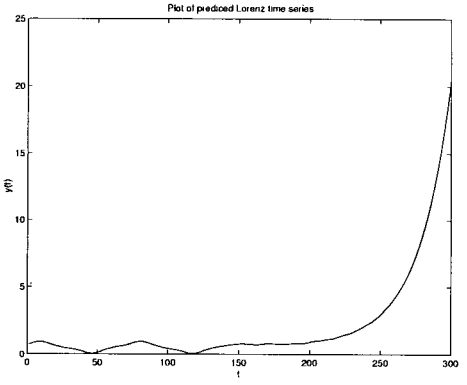

Figure 5.13: Plot of a prediction by the Sauer algorithm that displays divergent behaviour.

illustrated in figures 5.10, 5.11, 5.12 and 5.13.

5.2.3 Analysis of results for the Sauer algorithm.

The construction of the tables proceed in a similar fashion to that of the results tables for the TDNN except that only one predicted time series was obtained for each combination of the parameters.

Table 5.9 shows that to satisfy the behaviour criterion the dimension (regardless of delay) is best chosen to be 2 and the delay (regardless of dimension) chosen to be 6. This is not in accordance with the optimal embedding parameters.

However, the accuracy criterion (table 5.10) shows that prediction accuracy improves with an increase in the embedding dimension, which is in accordance with the concepts of a minimal embedding dimension. The embedding delay that corresponds with the smallest value for the MSE is 10. As was argued earlier, values for the time delay in an interval around 9 are acceptable.

Dimension	Time delay						
	6	7	8	9	10	11	mean
2	**0.031**	0.032	**0.030**	**0.028**	(**0.023**)	**0.025**	0.028
3	0.038	(**0.029**)	0.033	0.038	0.035	0.046	0.037
4	0.041	(0.029)	0.043	0.081	0.086	0.091	0.062
5	(0.046)	0.077	0.093	0.075	0.102	0.110	0.084
6	(0.091)	0.097	0.099	0.096	0.120	0.128	0.110
mean	0.049	0.053	0.060	0.064	0.073	0.080	

Table 5.9: Average MSE values for Sauer algorithm predictions using the behaviour criterion.

Dimension	Time delay						mean
	6	7	8	9	10	11	
2	(0.015)	0.016	0.017	0.017	0.018	0.018	0.017
3	(0.016)	0.018	0.017	0.017	0.017	(0.016)	0.017
4	0.016	0.015	0.017	0.016	0.012	(**0.011**)	0.014
5	(0.015)	0.015	(**0.010**)	0.014	(0.010)	0.012	0.012
6	**0.014**	**0.012**	0.011	**0.010**	(**0.009**)	**0.011**	0.011
mean	0.015	0.015	0.014	0.015	0.013	0.014	

Table 5.10: Average MSE values for Sauer algorithm predictions using the accuracy criterion.

A noteworthy observation arose by calculating the MSE for the full 300 points in each prediction. It should be noted that of the 5850 simulations arising from the combination of parameters only the best 4680 (80%) simulations were chosen to avoid the divergent predictions (figure 5.13) skewing the MSE values.

Dimension	Time delay					
	6	7	8	9	10	11
2	0.84	0.81	0.79	0.77	0.73	(0.71)
3	0.78	0.78	0.77	0.73	0.72	(0.70)
4	0.76	0.75	0.72	**0.69**	(**0.67**)	**0.68**
5	0.72	0.70	(**0.69**)	0.71	0.71	0.71
6	**0.70**	(**0.70**)	0.71	**0.71**	0.73	0.73

Table 5.11: Average MSE values for dimension versus delay.

Dimension	Spanning dimension				
	2	3	4	5	6
2	*0.78*	-	-	-	-
3	0.73	*0.77*	-	-	-
4	0.70	*0.71*	*0.71*	-	-
5	0.70	0.70	0.70	*0.74*	-
6	0.70	0.70	0.70	0.75	*0.75*

Table 5.12: Average MSE values for dimension versus spanning dimension. (Values in italics indicate maximum values for a row.)

Table 5.11 shows the average MSE values for different combinations of dimension and delay. The minimum MSE value for each delay occurs when the product of the dimension and delay is close or equal to 40. Similarly, minimum MSE value for each dimension occurs when the product of the dimension and delay is as close as

possible to 40.

It can be shown that the quasi-period of the Lorenz time series used for the predictions is approximately 40.

The product of the dimension and delay describes the length of the time series used in embedding one point. The Sauer algorithm thus predicts most accurately over a large prediction window when the product of the dimension and delay is equal to pseudo period of the time series in question.

Table 5.12 shows that a choice of the spanning dimension not equal to the embedding dimension will be adequate. Experiments for the optimal number of neighbours showed no best choice for the tested values.

Chapter 6

Conclusion

This thesis implemented and evaluated two different algorithms used for the prediction of deterministic chaotic time series. Both algorithms were evaluated on the basis of two criteria; prediction accuracy and behavioural accuracy.

The Lorenz time series was used because of its popularity as a test case for prediction algorithms.

Three questions were initially posed; (1) which combinations of embedding dimension and time lag are the best for prediction accuracy and for capturing of the dynamics, (2) are the two criteria satisfied by the same combination of test parameters and (3) are the best combinations of parameters independent of the choice of the algorithm?

In terms of the first question, results for the TDNN algorithm showed that embedding dimensions greater than values obtained for the minimum embedding dimension from the false neighbour heuristic provided better prediction accuracy. Similarly, the behavioural criterion was also best satisfied for values of the embedding dimension greater than the minimum embedding dimension.

Experiments performed with hidden context layers in the TDNN indicated that although the prediction accuracy showed a marginal improvement, the behavioural accuracy was reduced.

Results thus showed that values for the time delay lying in a small interval around the optimal time delay gave acceptable prediction and behavioural accuracy for the TDNN algorithm without hidden context layers, but that with the inclusion of hidden context layers the correct time delay could not be reliably ascertained for behavioural accuracy.

Predictions based on the Sauer algorithm provide a different result. The parameters found to satisfy the behavioural criterion was the combination of the smallest embedding dimension and time delay tested. This is in direct contrast to the expected optimal embedding dimensions.

However, the accuracy criterion is best satisfied by values for the embedding dimension greater than the minimal embedding dimension and time delays in an interval around the optimal time delay.

An interesting result was obtained for the Sauer algorithm prediction of a large number of points. It was found that for such long predictions the optimal embedding parameters were products of embedding dimension and time delay approximately equal to the quasi-period of the Lorenz attractor. This result could not be duplicated with the TDNN algorithm.

As would already have been noted, that bar the TDNN algorithm without hidden context layers, the two criteria are not consistently satisfied by the same combination of embedding dimension and time delay for either algorithm.

The final question, being whether the best parameters are independent of the choice of algorithm, is confirmed for the accuracy criterion but not for the behaviour criterion.

Future research resulting from this study could include a more rigorous means of comparing a time series and its prediction to better assess the behaviour criterion. A refinement of Sauer's algorithm could entail an adaptive neighbour selection method and an adaptive spanning dimension method derived from eigenvalue observations. The scope for experimentation with neural networks is vast, with various different forms of network architectures and training methodologies as subjects for investigation.

Appendix A

Time series scaling functions

Sauer's algorithm (chapter 3) is independent of the scaling of the time series values being predicted. Neural networks (chapter 4) do however require the application of specific scaling methods in order to obtain meaningful output from the network. To scale the data for the training set a method of weighting can be used, where Ψ is the scaled data series from Γ.

$$\Psi_j = \frac{\Gamma_j - \min(\Gamma)}{\max(\Gamma) - \min(\Gamma)} \tag{A.1}$$

The weighting function (A.1) scales the time series between 0 and 1. A variation on this binary form of scaling is the logarithmic scaling function,

$$\Psi_j = \frac{\log \Gamma_j - \log\left(\min(\Gamma)\right)}{\log\left(\max(\Gamma)\right) - \log\left(\min(\Gamma)\right)} \tag{A.2}$$

To scale the input between -1 and 1 the following function can be used,

$$\Psi_j = \frac{s}{\max\left(|\max(s)|, |\min(s)|\right)} \tag{A.3}$$

where

$$s = \frac{\Gamma_j - \overline{\Gamma}}{\sigma_\Gamma} \tag{A.4}$$

where σ_Γ and $\overline{\Gamma}$ are respectively the standard deviation and mean of the time series Γ.

For this thesis, all data was scaled with (A.1).

Appendix B

Sauer algorithm Matlab code

B.1 Main program

```
function [Predicted]=Sauer(EmbedDim,TimeDelay,NrNearestNeighbours,LastKnownPointIndex, ...
   NrPredictedPoints,SpanDim)

load nlor.mat;

Points=LastKnownPointIndex+NrPredictedPoints;

[TrainingSet,TargetSet]=EmbedTimeSeries(LorenzTimeSeries,EmbedDim,TimeDelay,Points, ...
   LastKnownPointIndex);

for PredPoint=(LastKnownPointIndex):(Points-1)
    TrueNeighbours=NeighbourSearch(EmbedDim,TimeDelay,PredPoint,TrainingSet,NrNearestNeighbours, ...
       LastKnownPointIndex);

    CentreOfMass=COM(EmbedDim,NrNearestNeighbours,TrueNeighbours,TrainingSet);

    [SpanningMatrix,DisplacementMatrix]=SingValDec(TrueNeighbours,CentreOfMass,TrainingSet, ...
       SpanDim,NrNearestNeighbours);

    ProjectedMatrix=Project(SpanningMatrix,DisplacementMatrix,NrNearestNeighbours);

    [Coefficients,Constants]=LinearRegression(LorenzTimeSeries,TrueNeighbours, ...
       NrNearestNeighbours,ProjectedMatrix,EmbedDim,TimeDelay);

    TargetSet = EstimateTargetPoint(EmbedDim,TimeDelay,PredPoint,TargetSet,Coefficients, ...
       Constants,CentreOfMass,SpanningMatrix);
end

Predicted=TargetSet(LastKnownPointIndex+1:Points,1);
```

B.2 Embedding the time series

```
function [TrainingSet,TargetSet]=EmbedTimeSeries(LorenzTimeSeries,EmbedDim,TimeDelay,Points, ...
   LastKnownPointIndex)

c=1;
while c<=Points;
    for i=1:EmbedDim
        TargetSet(c,(EmbedDim+1)-i) = LorenzTimeSeries((c+(i-1)*TimeDelay),1);
    end
    c = c+1;
end

TrainingSet(1:(LastKnownPointIndex),:)=TargetSet(1:(LastKnownPointIndex),:);
```

B.3 Finding nearest neighbours

```
function TrueNeighbours=NeighbourSearch(EmbedDim,TimeDelay,PredPoint,TrainingSet,NrNearestNeighbours, ...
   LastKnownPointIndex)

for i=1:LastKnownPointIndex-1-((EmbedDim-1)*TimeDelay+1)
    EuclideanDistance(i)=norm(TrainingSet(PredPoint,:)-TrainingSet(i,:));
end

for nu=1:NrNearestNeighbours
    [value index]=min(EuclideanDistance);
    TrueNeighbours(nu,:)=[value index];
    dstm(max(1,index-EmbedDim):index+EmbedDim)=max(EuclideanDistance);
end
```

B.4 Calculating the centre of mass of the nearest neighbours

```
function CentreOfMass=COM(EmbedDim,NrNearestNeighbours,TrueNeighbours,TrainingSet)

for i=1:EmbedDim
    numerator=0;
    denominator=0;
    for j=1:NrNearestNeighbours
       numerator=numerator+(1-(1/2)*(TrueNeighbours(j,1)/TrueNeighbours(NrNearestNeighbours,1))^2)^(3)* ...
          TrainingSet(TrueNeighbours(j,2),i);
       denominator=denominator+(1-(1/2)*(TrueNeighbours(j,1)/TrueNeighbours(NrNearestNeighbours,1))^2)^(3);
    end
    CentreOfMass(1,i)=numerator/denominator;
end
```

B.5 Singular value decomposition and the spanning matrix

```
function [SpanningMatrix,DisplacementMatrix]=SingValDec(TrueNeighbours,CentreOfMass,TrainingSet,SpanDim, ...
   NrNearestNeighbours)

for i=1:NrNearestNeighbours
    DisplacementMatrix(i,:)=TrainingSet(TrueNeighbours(i,2),:)-CentreOfMass;
end

[U,S,V]=svd(DisplacementMatrix);
SpanningMatrix(:,1:SpanDim)=V(:,1:SpanDim);
```

B.6 Projecting the displacement matrix

```
function ProjectedMatrix=Project(SpanningMatrix,DisplacementMatrix,NrNearestNeighbours)

ThetaMatrix=SpanningMatrix*inv(SpanningMatrix'*SpanningMatrix)*SpanningMatrix';
for i=1:NrNearestNeighbours
    prja(:,i)=ThetaMatrix*DisplacementMatrix(i,:)';
end

ProjectedMatrix=prja';
```

B.7 Performing the linear regression

```
function [Coefficients,Constants]=LinearRegression(LorenzTimeSeries,TrueNeighbours, ...
   NrNearestNeighbours,ProjectedMatrix,EmbedDim,TimeDelay)

for i=1:NrNearestNeighbours
    X(i,:)=LorenzTimeSeries(TrueNeighbours(i,2)+(EmbedDim-1)*TimeDelay+1,:);
end

[r,c]=size(ProjectedMatrix);
Pro=[ProjectedMatrix ones(r,1)];

%perform linear regression
Coefficients=Pro\X;

Constants=Coefficients(length(Coefficients),:);
Coefficients(length(Coefficients),:)=[];
```

B.8 Estimating the target point

```
function TargetSet = EstimateTargetPoint(EmbedDim,TimeDelay,PredPoint,TargetSet,Coefficients, ...
   Constants,CentreOfMass,SpanningMatrix)

for i=2:EmbedDim
    TargetSet(PredPoint+1,i)=TargetSet(PredPoint-TimeDelay+1,i-1);
end
ProjectedMatrixa=SpanningMatrix*inv(SpanningMatrix'*SpanningMatrix)*SpanningMatrix';
nxtpnt=ProjectedMatrixa*(TargetSet(PredPoint,:)-CentreOfMass)';
TargetSet(PredPoint+1,1)=nxtpnt'*Coefficients+Constants';
```

Appendix C

TDNN algorithm Matlab code

C.1 TDNN training code

```
function [RMSError]=TDNNTrain(NrTrainingPatterns,Epochs,NrInputNodes,NrHiddenNodes, ...
   NrInputNodeTaps,NrOutputNodes,NrHiddenNodeTaps,EmbedDim,TimeDelay)

RootMeanSquareError=[];
Momentum=0.75;

%Load data

load TrainingPoints
load LearningRate

%Start epoch

while size(RootMeanSquareError,2)<Epochs
    load Weights
    ErrorTemp=0;

    InputLayer=zeros(1,(NrInputNodes*(NrInputNodeTaps+1)));
    HiddenLayer=zeros(1,(NrHiddenNodes*(NrHiddenNodeTaps+1)));
    OutputLayer=zeros(1,NrOutputNodes);

    InputLayer(NrInputNodes*(NrInputNodeTaps+1)+1)=-1;
    HiddenLayer(NrHiddenNodes*(NrHiddenNodeTaps+1)+1)=-1;

    %Tap preparation

    for pttn=1:(max(NrInputNodeTaps,NrHiddenNodeTaps)+1)
        InputLayer(1:NrInputNodes)=InputPattern(pttn,:);

        nrn_tmp1=(Weight_HiddenLayer*InputLayer')';
        HiddenLayer(1:NrHiddenNodes)=tanh(nrn_tmp1);

        nrn_tmp2=(Weight_OutputLayer*HiddenLayer')';
        OutputLayer(1:NrOutputNodes)=1./(1+exp(-1*nrn_tmp2));

        for n=NrInputNodeTaps+1:-1:2
            InputLayer((n-1)*NrInputNodes+1:n*NrInputNodes)=0.1*InputLayer((n-2)*NrInputNodes+1: ...
               (n-1)*NrInputNodes);
        end
        for m=NrHiddenNodeTaps+1:-1:2
            HiddenLayer((m-1)*NrHiddenNodes+1:m*NrHiddenNodes)=0.1*HiddenLayer((m-2)* ...
               NrHiddenNodes+1:(m-1)*NrHiddenNodes);
        end

    end

    %Forward propagation

    h_Weight_OutputLayer=zeros(NrOutputNodes,((NrHiddenNodeTaps+1)*NrHiddenNodes+1));
    h_Weight_HiddenLayer=zeros(NrHiddenNodes,((NrInputNodeTaps+1)*NrInputNodes+1));

    for pttn=(max(NrInputNodeTaps,NrHiddenNodeTaps)+2):NrTrainingPatterns

        InputLayer(1:NrInputNodes)=InputPattern(pttn,:);

        nrn_tmp1=(Weight_HiddenLayer*InputLayer')';
```

```
        HiddenLayer(1:NrHiddenNodes)=tanh(nrn_tmp1);

        nrn_tmp2=(Weight_OutputLayer*HiddenLayer')';
        OutputLayer(1:NrOutputNodes)=1./(1+exp(-1*nrn_tmp2));

        %Error calculation

        ErrorTemp=ErrorTemp+(OutputPattern(pttn,:)-OutputLayer)*(OutputPattern(pttn,:)-OutputLayer)';

        ErrorOutputLayer=(OutputPattern(pttn,:)-OutputLayer)*diag(diag((1-OutputLayer)'*OutputLayer));

        ErrorSumHidden=sum(ErrorOutputLayer*Weight_OutputLayer);
        ErrorHiddenLayer=(1-HiddenLayer.^2)*ErrorSumHidden;

        %Back propagation

        t_Weight_OutputLayer=Weight_OutputLayer;
        Weight_OutputLayer=Weight_OutputLayer+Momentum(size(Momentum,2))*h_Weight_OutputLayer+ ...
           LearnRate(size(LearnRate,2))*ErrorOutputLayer'*HiddenLayer;
        h_Weight_OutputLayer=Weight_OutputLayer-t_Weight_OutputLayer;

        t_Weight_HiddenLayer=Weight_HiddenLayer;
        for j=1:NrHiddenNodes
            Weight_HiddenLayer(j,:)=Weight_HiddenLayer(j,:)+Momentum(size(Momentum,2))* ...
               h_Weight_HiddenLayer(j,:)+LearnRate(size(LearnRate,2))*ErrorHiddenLayer(j)* ...
               InputLayer;
        end
        h_Weight_HiddenLayer=Weight_HiddenLayer-t_Weight_HiddenLayer;

        %Update neurons

        for n=NrInputNodeTaps+1:-1:2
            InputLayer((n-1)*NrInputNodes+1:n*NrInputNodes)=0.1*InputLayer((n-2)*NrInputNodes+1: ...
               (n-1)*NrInputNodes);
        end

        for m=NrHiddenNodeTaps+1:-1:2
            HiddenLayer((m-1)*NrHiddenNodes+1:m*NrHiddenNodes)=0.1*HiddenLayer((m-2)* ...
               NrHiddenNodes+1:(m-1)*NrHiddenNodes);
        end

    end

    %RMS Error

    TDNNEvaluate(NrTrainingPatterns,Epochs,NrInputNodes,NrHiddenNodes,NrOutputNodes, ...
       NrInputNodeTaps,NrHiddenNodeTaps,Weight_HiddenLayer,Weight_OutputLayer,EmbedDim,TimeDelay);

    RootMeanSquareError=[RootMeanSquareError (1/NrTrainingPatterns)*(1/NrOutputNodes)* ...
       sqrt(ErrorTemp)];

    if size(RootMeanSquareError,2)>5
        Differences= ...
           diff(RootMeanSquareError(size(RootMeanSquareError,2)-2:size(RootMeanSquareError,2)));
        GradientNegative=length(find((sign(Differences))<0));
        if GradientNegative==2
            LearnRate(size(LearnRate,2)+1)=LearnRate(size(LearnRate,2))*1.05;
            if Momentum(size(Momentum,2))<=0.98
                Momentum(size(Momentum,2)+1)=Momentum(size(Momentum,2))+0.01;
            end
            save Weights Weight_HiddenLayer Weight_OutputLayer

        elseif GradientNegative==0
            LearnRate(size(LearnRate,2)+1)=LearnRate(size(LearnRate,2))*0.7;
            if Momentum(size(Momentum,2))>=0.02
                Momentum(size(Momentum,2)+1)=Momentum(size(Momentum,2))-0.01;
            end
        else
            save Weights Weight_HiddenLayer Weight_OutputLayer
        end
    else
        save Weights Weight_HiddenLayer Weight_OutputLayer
    end
end
RMSError=RootMeanSquareError;
```

C.2 TDNN evaluation function

```
function TDNNEvaluate(NrTrainingPatterns,Epoch,NrInputNodes,NrHiddenNodes,NrOutputNodes, ...
   NrInputNodeTaps,NrHiddenNodeTaps,Weight_HiddenLayer,Weight_OutputLayer,EmbedDim,TimeDelay)
```

```
NrPredictionPoints=300;

%Load data

load PredictionTrainingPoints
load TimeSeries

%Epoch start

InputLayer=zeros(1,(NrInputNodes*(NrInputNodeTaps+1)));
HiddenLayer=zeros(1,(NrHiddenNodes*(NrHiddenNodeTaps+1)));
OutputLayer=zeros(1,NrOutputNodes);

InputLayer(NrInputNodes*(NrInputNodeTaps+1)+1)=-1;
HiddenLayer(NrHiddenNodes*(NrHiddenNodeTaps+1)+1)=-1;

xpdat=[];

%Forward propagation

for pttn=1:(max(Itap,Jtap)+1)
        InputLayer(1:NrInputNodes)=InputPattern(pttn,:);

        nrn_tmp1=(Weight_HiddenLayer*InputLayer')';
        HiddenLayer(1:NrHiddenNodes)=tanh(nrn_tmp1);

        nrn_tmp2=(Weight_OutputLayer*HiddenLayer')';
        OutputLayer(1:NrOutputNodes)=1./(1+exp(-1*nrn_tmp2));

        for n=NrInputNodeTaps+1:-1:2
            InputLayer((n-1)*NrInputNodes+1:n*NrInputNodes)=0.1*InputLayer((n-2)*NrInputNodes+1: ...
               (n-1)*NrInputNodes);
        end

        for m=NrHiddenNodeTaps+1:-1:2
            HiddenLayer((m-1)*NrHiddenNodes+1:m*NrHiddenNodes)=0.1*HiddenLayer((m-2)* ...
               NrHiddenNodes+1:(m-1)*NrHiddenNodes);
        end

end

% Prediction phase

for pttn=1+(max(Itap,Jtap)+1):NrPredictionPoints+(max(Itap,Jtap)+1)
        InputLayer(1:NrInputNodes)=InputPattern(pttn,:);

        nrn_tmp1=(Weight_HiddenLayer*InputLayer')';
        HiddenLayer(1:NrHiddenNodes)=tanh(nrn_tmp1);

        nrn_tmp2=(Weight_OutputLayer*HiddenLayer')';
        OutputLayer(1:NrOutputNodes)=1./(1+exp(-1*nrn_tmp2));

        for n=NrInputNodeTaps+1:-1:2
            InputLayer((n-1)*NrInputNodes+1:n*NrInputNodes)=0.1*InputLayer((n-2)*NrInputNodes+ ...
            1:(n-1)*NrInputNodes);
        end

        for m=NrHiddenNodeTaps+1:-1:2
            HiddenLayer((m-1)*NrHiddenNodes+1:m*NrHiddenNodes)=0.1*HiddenLayer((m-2)* ...
            NrHiddenNodes+1:(m-1)*NrHiddenNodes);
        end

    %Insert predicted points

    InputPattern(pttn+1,1:NrOutputNodes)=OutputLayer;
    for i=2:NrInputNodes
        InputPattern(pttn+1,i)=InputPattern(pttn,i-1);
    end

    PredictedOutput=[PredictedOutput;OutputLayer];
end

load realdata
eval(['rH' num2str(EmbedDim) num2str(TimeDelay) '= OutputPattern(1+(max(NrInputNodeTaps, ...
   NrHiddenNodeTaps)+1):NrPredictionPoints+(max(NrInputNodeTaps,NrHiddenNodeTaps)+1));']);
save realdata rH*

load PredictedOutput
NeuralPredictedData=[NeuralPredictedData;PredictedOutput'];
save PredictedOutput NeuralPredictedData
```

C.3 Weight initialization function

```
function TDNNInitialize(EmbedDim,TimeDelay,NrTrainingPatterns,Epoch,NrInputNodes,NrHiddenNodes, ...
   NrInputNodeTaps,NrHiddenNodeTaps)

load LorenzTimeSeries.mat

NrOutputNodes=1;
NrPredictionPoints=300;

data=[];
for d=1:19000
    data=[data transX((d-1)*TimeDelay+1)];
end

for i=1:2000
    ts(i,:)=data(i:i+EmbedDim-1);
end
TimeSeries=fliplr(ts);

InputPattern(1:NrTrainingPatterns,:)=timesrs(1:NrTrainingPatterns,:);
OutputPattern(1:NrTrainingPatterns,1:NrOutputNodes)=timesrs(NrOutputNodes+1:NrOutputNodes+ ...
   NrTrainingPatterns,1:NrOutputNodes);

save TrainingPoints InputPattern OutputPattern
clear InputPattern OutputPattern

InputPattern(1:NrPredictionPoints+(max(NrInputNodeTaps,
   NrHiddenNodeTaps)+1),:)=timesrs(NrTrainingPatterns+1:NrTrainingPatterns+NrPredictionPoints+ ...
      (max(NrInputNodeTaps,NrHiddenNodeTaps)+1),:);
OutputPattern(1:NrPredictionPoints+(max(NrInputNodeTaps,
   NrHiddenNodeTaps)+1),1:NrOutputNodes)=timesrs(NrOutputNodes+NrTrainingPatterns+1: ...
   NrOutputNodes+NrTrainingPatterns+NrPredictionPoints+(max(NrInputNodeTaps,NrHiddenNodeTaps) ...
   +1),1:NrOutputNodes);

save PredictionTrainingPoints InputPattern OutputPattern

nram1=(rand(NrHiddenNodes,((NrInputNodeTaps+1)*NrInputNodes+1))/10);
nram2=(rand(NrHiddenNodes,((NrInputNodeTaps+1)*NrInputNodes+1))/10);
nram3=(rand(NrOutputNodes,((NrHiddenNodeTaps+1)*NrHiddenNodes+1))/10);
nram4=(rand(NrOutputNodes,((NrHiddenNodeTaps+1)*NrHiddenNodes+1))/10);

Weight_HiddenLayer=nram1-nram2;
Weight_OutputLayer=nram3-nram4;

LearnRate=0.035;

save TimeSeries TimeSeries
save LearningRate LearnRate
save Weights Weight_OutputLayer Weight_HiddenLayer
```

Bibliography

[1] H.D.I. Abarbanel, M.E. Gilpin, and M. Rotenberg. *Analysis of observed chaotic data*. Institute for Nonlinear Science. Springer Verlag, New York, 1996.

[2] A.M. Albano, A. Passamante, T. Hediger, and M.E. Farrell. Using neural nets to look for chaos. *Physica D*, 58:1–9, 1992.

[3] R. Bakker, J.C. Schouten, C.L. Giles, F. Takens, and C.M. van den Bleek. Learning chaotic attractors by neural networks. *Neural Computation*, 12(10):2355–2383, 2000.

[4] N. Davey, S.P. Hunt, and R.J. Frank. Time series prediction and neural networks. *Journal of Intelligent and Robotic Systems*, 31:91–103, 2001.

[5] H. Demuth and M. Beale. *Neural network toolbox user's guide*. The MathWorks Inc., Natick, Massachusetts, 1998.

[6] J.L. Elman. Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning*, 7:195–225, 1991.

[7] J.B. Elsner. Predicting time series using a neural network as a method of distinguishing chaos from noise. In T. Kohonen, K. Mäkisara, O. Simula, and J. Kangas, editors, *Artificial neural networks*, pages 145–150. Elsevier Science Publishers, North-Holland, 1991.

[8] J.D. Farmer and J.J. Sidorowich. Predicting chaotic time series. *Physical Review Letters*, 59(9):845–848, 1987.

[9] A.M. Fraser and H.L. Swinney. Independent coordinates for strange attractors from mutual information. *Physical Review A*, 33(2):1134–1140, 1986.

[10] P. Grassberger and I. Procaccia. Characterization of strange attractors. *Physical Review Letters*, 50(5):346–349, 1983.

[11] R. Hegger, H. Kantz, and T. Schreiber. Practical implementation of nonlinear time series methods: The TISEAN package. *Chaos*, 9(2):413–435, 1999.

[12] H. Kantz and T. Schreiber. *Nonlinear time series analysis*. Cambridge nonlinear science series 7. Cambridge University Press, Cambridge, 1997.

[13] M.B. Kennel and H.D.I. Abarbanel. False neighbors and false strands: A reliable minimum embedding dimension algorithm. *INLS Preprint*, [Online] http://inls.ucsd.edu/ftp/pub/inls-ucsd/false_strands.tar.Z. cited 2002, Nov 26.

[14] M.B. Kennel, R. Brown, and H.D.I. Abarbanel. Determining embedding dimension for phase space reconstruction using the method of false nearest neighbors. *Physical Review A*, 45:3403–3411, 1992.

[15] E. Kreyszig. *Introductory mathematical statistics*. John Wiley and Sons, New York, 1970.

[16] E.N. Lorenz. Deterministic non-periodic flow. *Journal of Atmospheric Science*, 20:130–141, 1963.

[17] M.C. Mozer. Neural net architectures for temporal sequence processing. In A.S. Weigend and N.A. Gershenfeld, editors, *Time series prediction: forecasting the future and understanding the past*, pages 243–264. Addison-Wesley, Readwood City, California, 1993.

[18] E. Ott, T. Sauer, and J.A. Yorke. *Coping with chaos: analysis of chaotic data and the exploitation of chaotic systems*. Wiley series in nonlinear science. John Wiley and Sons, New York, 1994.

[19] N.H. Packard, J.P. Crutchfield, J.D. Farmer, and R.S. Shaw. Geometry from a time series. *Physical Review Letters*, 45(9):712–716, 1980.

[20] T. Sauer, J.A. Yorke, and M. Casdagli. Embedology. *Journal of Statistical Physics*, 65(3-4):579–616, 1991.

[21] C.G. Schroer, T. Sauer, E. Ott, and J.A. Yorke. Predicting chaos most of the time from embeddings with self-intersections. *Physical Review Letters*, 80(7):1410–1413, 1998.

[22] M. Smith. *Neural networks for statistical modelling*. Van Nostrand Reinhold, New York, 1993.

[23] C. Sparrow. *The Lorenz equations: bifurcations, chaos and strange attractors*, volume 41 of *Applied Mathematical Sciences*. Springer-Verlag, New York, 1982.

[24] G. Strang. *Introduction to linear algebra*. Wellesley-Cambridge Press, Wellesley, Massachusetts, 1993.

[25] K. Swingler. *Applying neural networks a practical guide*. Academic Press, London, 1996.

[26] V. Tresp and R. Hofmann. Nonlinear time-series prediction with missing and noisy data. *Neural Computation*, 10:731–747, 1998.

[27] A.S. Weigend and N.A. Gershenfeld. *Time series prediction: forecasting the future and understanding the past*. Addison-Wesley, Reading, Massachusetts, 1994.

[28] J.M. Zurada. *Introduction to artificial neural systems*. West Publishing Company, St. Paul, Minnesota, 1992.