Asymptotically Stable Neural Networks for Identification of Nonlinear Dynamic Systems

by

Chris M. Jubien

B.A.Sc., University of Waterloo, Waterloo Ontario, 1990.

A thesis submitted in partial fulfillment of the requirements of the degree

MASTER OF APPLIED SCIENCE

in the Department of
Electrical and Computer Engineering,
University of Victoria,
Victoria, BC Canada

We accept this thesis as conforming to the required standard

[redacted]

Dr. N. Dimopoulos, Supervisor (Dep't of Electrical and Computer Engineering)

[redacted]

Dr. W.S. Lu, Dep't Member (Dep't of Electrical and Computer Engineering)

[redacted]

Dr. G. McLean, Outside Member (Dep't of Mechanical Engineering)

[redacted]

Dr. Y. Stepanenko, External Examiner (Dep't of Mechanical Engineering)

Supervisor : Dr. N. Dimopoulos

Abstract

This thesis deals with the use of stable neural networks for identification of nonlinear dynamic systems.

Neural Networks have been in increasing use. Lately, they have been applied to identification of nonlinear systems. Neural networks have the advantage that they are extremely flexible and, once trained, provide accurate and fast modeling of complex systems. Proofs are available which show that neural networks can implement systems of arbitrary complexity.

Here, a novel approach to system identification is used. A class of recurrent neural networks which have been proven to be asymptotically stable are the basis for identification. Equations for training these neural networks to model the complex behaviour that nonlinear systems exhibit are developed. Computer simulations are used to test the theory. In a simulated system, the neural network is found to provide good modeling. The equations are also applied to train a neural network to model the dynamic behaviour of a boat and a PUMA-560 two-link robot.

Examiners:

[redacted]

Dr. N. Dimopoulos, Supervisor (Dep't of Electrical and Computer Engineering)

[redacted]

Dr. W.S. Lu, Dep't Member (Dep't of Electrical and Computer Engineering)

[redacted]

Dr. G. McLean, Outside Member (Dep't of Mechanical Engineering)

[redacted]

Dr. Y. Stepanenko, External Examiner (Dep't of Mechanical Engineering)

Table of Contents

List of Figures

Acknowledgments

This thesis would not have been possible without the assistance of lots of people. I would like to thank some of the most important ones:

Dr. Dimopoulos, for his help and advice in making this thesis;

Dr. Lu, for having time to listen to ideas;

Mile Erlic, for help with robotics;

Alex Andekian, for help with the section on boat identification;

Steve Rayson and Henry Weinhard, for their advice on programming;

Samantha, for helping me finish this thesis;

Bernie, Bart, Rocky and Molly, for many interesting discussions;

And Mom, Dad, and Gary, for all the support they've always given me.

To all these people, a hearty thanks.

For Casey

Introduction

This thesis is entitled "Asymptotically Stable Neural Networks for Identification of Nonlinear Dynamic Systems". The purpose of this thesis is to present some recent work which unites the fields of neural networks and nonlinear system identification.

In recent years, much has been accomplished in the field of neural networks. Investigations have been made as to what neural networks can do. This thesis continues this investigation process, but it will be pay more attention to known theoretical considerations than most neural network research. The most important consequence of this is that the most fundamental idea of control theory, which is the concept that the priority of a controller is to provide stability, will be observed at all times. This means that the dynamic systems which are created by the neural networks used here will be proven to be asymptotically stable.

This approach causes some difficulty since although a class of neural networks exists which is known to be asymptotically stable, no method exists in prior work for training such neural networks to have the kind of complex behaviour which is exhibited by nonlinear systems. Much of the work here will be concentrated on developing useful methods for training such neural networks.

Motivation will also be given for this approach by investigating nonlinear systems. It is common in the liteature to assume that a certain neural network is stable since it has been tested using methods common to linear system theory and appears to be stable. In this thesis, nonlinear systems which appear to pass tests for linear system stability, but are actually unstable, will be analyzed. The results will provide further motivation for the necessity of using neural networks which are known to be stable in critical applications.

To prove the utility of the neural networks which are discussed here, simulations are used to test system identification ability. Some of theses simulations are abstract computer

simulations, but some involve actual data in the identification of a robot. This is intended to lend credence to the training methods which are developed here.

The structure of this thesis is as follows. Chapter 1 is intended to provide the background information on linear systems, nonlinear systems, and neural networks which is needed to understand the discussion in the rest of the thesis. Chapter 2 discusses identification in the context of this thesis. Chapter 3 presents the theoretical work which is used to develop a stable neural network suitable for identification of nonlinear systems. Chapter 4 presents the results of simulations which use the developments of chapter 3. Chapter 5 provides the conclusions of this thesis.

Chapter 1: Background

1.0 Linear Systems

1.0.0 Linear Dynamic Systems

The main purpose of this section is to discuss the definition of a linear dynamic system. Linear systems are discussed throughout this thesis, and are used to give intuitive insight into the operation of nonlinear systems. Both continuous time and discrete time dynamic systems will be discussed here. For a thorough discussion of the subjects mentioned here, consult [9],[15], or [16].

Consider a continuous time differential equation which may be considered to have an input $u(t)$ and an output $y(t)$. A general differential equation will have the form

$$f\left(t, y(t), y'(t), y''(t), \ldots, y^{(n)}(t), u(t), u'(t), u''(t), \ldots, u^{(m)}(t)\right) = k \quad (1.1)$$

where $k \in R$. Although this form is the most general one available, it is very difficult to make any useful generalizations about this equation. However, many systems and processes in real life give rise to differential equations which may be well approximated by the form

$$a_0 y(t) + a_1 y'(t) + \cdots + a_n y^{(n)}(t) - b_0 u(t) - b_1 u'(t) - \cdots - b_m u^{(m)}(t) = 0 \quad (1.2)$$

where the a_i's and b_i's are real valued constants. Equations of this form are called linear, time-invariant differential equations. Of course, linear differential equations obey the principles of superposition and scaling of response equal to scaling of inputs[16].

In a real system, the most fundamental question which may be asked is, what happens to the system in the long term? Specifically, it is important to know if the output will continue to grow in magnitude as time evolves, eventually reaching some critical value and having catastrophic results. This is known as the stability question. A real valued

function $f(t)$ is stable if there exists a finite valued number $\lambda > 0$ such that $|f(t)| \leq \lambda \quad \forall\, t \geq 0$. If the solution $y(t)$ of equation (1.1) is stable in this sense to *all* stable inputs $u(t)$ then the system is said to be bounded input, bounded output (BIBO) stable.

It is important to note that if a system is not BIBO stable, then it does not necessarily follow that *every* stable input will cause an unstable output, but only that *some* stable input will cause an unstable output. For instance, the zero input will cause a stable output for any linear system, namely the zero output. This is an important fact since it means that stability of a system cannot be tested merely by applying a stable input and watching to see if the output is stable.

1.0.1 Linear Transformations

Let V and W be vector spaces. The function $T:V \rightarrow W$ is called a **linear transformation** of V into W if the following two properties are true for all **u** and **v** in V and for any scalar c:

1.1 $T(\mathbf{u} + \mathbf{v}) = T(\mathbf{u}) + T(\mathbf{v})$ (superposition)

1.2 $T(c\mathbf{u}) = cT(\mathbf{u})$ (invariance under scaling) [p.310, reference 12]

The Laplace Transform, defined as

$$\mathcal{L}[x(t)] = X(s) = \int_0^\infty e^{-st} x(t)\,dt \tag{1.3}$$

is a linear transform. This transformation is used extensively in control theory. Its main use is in the analysis of linear, time-invariant differential equations since it can transform a differential equation into a polynomial equation. This greatly simplifies the analysis of the differential equation, especially with regards to stability.

For the vector space of discrete sequences, the z-transform is analogous to the Laplace Transform for continuous functions. The z-transform is a linear transform defined on a sequence $\{e(k)\}$ as

$$E(z) = \mathcal{Z}[\{e(k)\}] = \sum_{k=0}^{\infty} e(k) z^{-k} \tag{1.4}$$

As with the Laplace Transform, the z-transform can be used to turn difference equations into polynomial equations.

It is also well known that if

$$y(t) = \int_0^t h(t-\tau) x(\tau) d\tau \tag{1.5}$$

then

$$Y(s) = H(s) X(s) \tag{1.6}$$

For the proof of this, refer to [p.238 reference 16]. Similarly for discrete sequences, if

$$y(k) = \sum_{n=0}^{k} h(k-n) x(n) \tag{1.7}$$

then

$$Y(z) = H(z) X(z) \tag{1.8}$$

1.0.2 Transfer Functions

A transfer function is defined as the ratio of the input of a system to the resulting output, as expressed in the frequency domain. For discrete linear time invariant systems with input $Y(z)$ and output $U(z)$,

$$Y(z) = H(z)U(z) \tag{1.9}$$

$H(z)$ is called the transfer function of the system. The transfer function describes how the system transfers the input to the output. The value of this transfer function, in addition to facilitating the solution of the difference equation, is that it completely separates the reliance of the system behaviour on the input. This is to say that the dynamics of the system may be studied independently from those of the input. This is highly desirable since it was mentioned that the test for BIBO stability of a system is that the output must be stable for all stable inputs. If the transfer function allows for testing of the stability of the system separately from the stability of the input, then the difficulty of the testing job has been reduced. As will be seen in more detail in the following section on stability, it is a simple matter to say whether the transfer function is stable, and thus if the output will be stable for *all* stable inputs.

1.0.3 Stability of Linear Systems

It is well known that the transfer function $\frac{A(s)}{B(s)}$ will be stable if and only if all of its poles have negative real parts[8]. BIBO stability is concerned with the response of the system to stable inputs, and thus the poles of the input may all be assumed to have negative real parts. Thus, the BIBO stability of the system may be examined without any reliance on the input merely by testing the poles of the transfer function. Specifically, if the poles of the transfer function $H(s)$ all have negative real parts, then the system will be BIBO stable.

A similar result is available for discrete time systems. A discrete transfer function $H(z)$ will be stable provided that all of its poles have magnitude less than one, i.e., $|z_i| < 1$ for each of its poles. In general, the poles of a discrete transfer function may be complex, and thus this test for stability is sometimes referred to as "having the poles within the unit circle".

Graphs showing the stable and unstable regions for poles for continuous and discrete systems are shown in Figure 1.1.

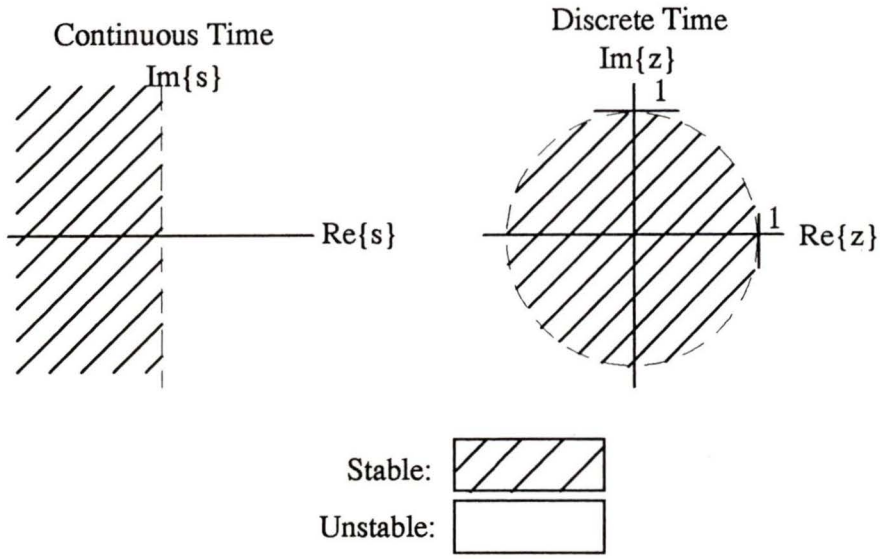

Figure 1.1: Stable and Unstable Pole Positions

1.0.4 State Space Representation

This section discusses an alternate way for representing a linear system. This is the state space method of representation. It is more general than the transfer function since state space models can be used to represent time-varying systems and multi-input, multi-output systems as well as the single-input single-output ones.

A state space representation takes a linear differential equation of order n and splits it into n first order differential equations. In general, the state space representation of an nth-order system with m inputs and k outputs will be of the form

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \\ \mathbf{y} &= \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u} \end{aligned} \tag{1.10}$$

where $\mathbf{x}$ is nX1, $\mathbf{A}$ is nXn, $\mathbf{B}$ is nXm, $\mathbf{u}$ is mX1, $\mathbf{y}$ is kX1, $\mathbf{C}$ is kXn, and $\mathbf{D}$ is kXm. For a time varying system, the matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{D}$ may be made time varying. In virtually all

real systems, there is some propagation lag from input to output, and thus the matrix **D**, which represents immediate or direct transfer of the input to the output, is the zero matrix. When **D** is zero, some of the calculations to be shown here are simplified slightly, and so it will be assumed that $\mathbf{D} = 0$.

It is a simple matter to switch from a state space representation of a linear time invariant system to a transfer function representation. In general,

$$\frac{a_{n-1}s^{n-1} + \cdots a_1 s + a_0}{s^n + b_{n-1}s^{n-1} + \cdots + b_0} \quad \Leftrightarrow$$

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ & & \vdots & & \\ 0 & 0 & 0 & \cdots & 1 \\ -b_0 & -b_1 & -b_2 & \cdots & -b_{n-1} \end{bmatrix}_{n\times n} \qquad \mathbf{B} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}_{n\times 1}$$

$$\mathbf{C} = \begin{bmatrix} a_0 & a_1 & a_2 & \cdots & a_{n-1} \end{bmatrix}_{1\times n} \tag{1.11}$$

It is possible to represent the same system with an infinite number of state space models through the use of similarity transforms. Let the matrix S be invertible, and let $\tilde{\mathbf{x}} = \mathbf{S}\mathbf{x}$, and therefore $\mathbf{x} = \mathbf{S}^{-1}\tilde{\mathbf{x}}$. From (1.20), if $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$, then premultiplying by S yields

$$\begin{aligned} \mathbf{S}\dot{\mathbf{x}} &= \mathbf{S}\mathbf{A}\mathbf{x} + \mathbf{S}\mathbf{B}\mathbf{u}, \quad \text{or} \\ \dot{\tilde{\mathbf{x}}} &= \mathbf{S}\mathbf{A}\mathbf{S}^{-1}\tilde{\mathbf{x}} + \mathbf{S}\mathbf{B}\mathbf{u} \end{aligned} \tag{1.12}$$

The output **y** is obtained as $\mathbf{y} = \mathbf{C}\mathbf{S}^{-1}\tilde{\mathbf{x}}$. Now letting $\tilde{\mathbf{A}} = \mathbf{S}\mathbf{A}\mathbf{S}^{-1}$, $\tilde{\mathbf{B}} = \mathbf{S}\mathbf{B}$, and $\tilde{\mathbf{C}} = \mathbf{C}\mathbf{S}^{-1}$, the equivalent state space system

$$\begin{aligned} \dot{\tilde{\mathbf{x}}} &= \tilde{\mathbf{A}}\tilde{\mathbf{x}} + \tilde{\mathbf{B}}\mathbf{u} \\ \mathbf{y} &= \tilde{\mathbf{C}}\tilde{\mathbf{x}} \end{aligned} \tag{1.13}$$

can be written. Note that this representation has the identical input-output relation as the original system. The transformation $\tilde{\mathbf{A}} = \mathbf{SAS}^{-1}$ is called a similarity transform. One important property of similar matrices is that they have the same eigenvalues.

The solution of a state space differential equation will be stable if and only if each eigenvalue of the system matrix A has a negative real part. This criterion is of course identical to the test of a transfer function, which had to have all of its poles with negative real parts for stability. In fact, the eigenvalues of a state space linear time invariant representation are always equal to the poles of the system's transfer function.

State space representations are also available for discrete time linear systems. In this case, the system is described as

$$\begin{aligned} \mathbf{x}(t+1) &= \mathbf{Ax}(t) + \mathbf{Bu}(t) \\ \mathbf{y}(t) &= \mathbf{Cx}(t) + \mathbf{Du}(t) \end{aligned} \tag{1.14}$$

It is also easy to switch from a discrete transfer function to a state space representation:

$$\frac{a_{n-1}z^{-(n-1)} + \cdots a_1 z^{-1} + a_0}{z^{-n} + b_{n-1}z^{-(n-1)} + \cdots + b_0} \quad \Leftrightarrow$$

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ & & \vdots & & \\ 0 & 0 & 0 & \cdots & 1 \\ -b_0 & -b_1 & -b_2 & \cdots & -b_{n-1} \end{bmatrix}_{n\times n} \qquad \mathbf{B} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}_{n\times 1}$$

$$\mathbf{C} = \begin{bmatrix} a_0 & a_1 & a_2 & \cdots & a_{n-1} \end{bmatrix}_{1\times n} \tag{1.15}$$

As is the case with continuous systems, the poles of the discrete transfer function correspond to the eigenvalues of the system matrix of the state space representation. As was discussed in the section on stability, discrete systems must have all their poles within

the unit circle to ensure stability, and thus all eigenvalues must be less than unity magnitude.

1.2 Nonlinear Dynamic Systems

1.2.0 Background for Nonlinear Differential Equations

This section discusses some basic mathematical definitions which are required for some of the material discussed in this Chapter and in Chapter 2.

Definition : C^0, the set of continuous functions, is defined as:

Let $C^0 = \{f : R^n \to R^n | f \text{ is continuous}\}$

Definition : C^1, the set of continuous differntiable functions, is defined as:

Let $C^1 = \{f : R^n \to R^n | f \in C^0, f' \in C^0\}$

Definition : Flow of a differential equation

Let $f \in C^1, f : R^n \to R^n$. Let $x(t)$ be the solution to the differential equation $x'(t) = f(x), x(0) = x_0$. Then the flow determined by f is the n-parameter set of all possible such solutions.

The flow is written as an operator which acts on an initial value, i.e., $g^t x(0)$. The flow may be thought of as describing the evolution in time of the physical system which f represents for all possible initial values. A pictorial representation of the flow, where possible, is perhaps the most meaningful way it may be represented. For instance, it is known that for n=2 and $f(x) = \begin{bmatrix} x_1 \\ -x_2 \end{bmatrix}$, the flow for $x_1(0), x_2(0) > 0$ is as shown in Figure 1.2.

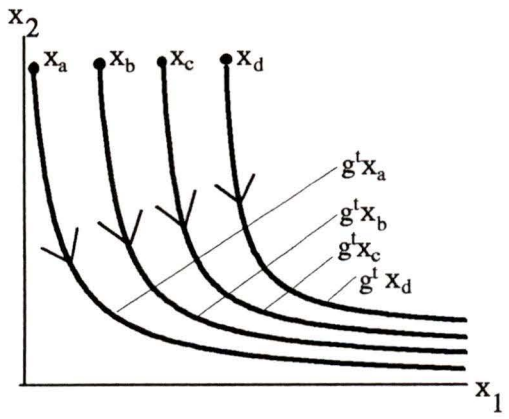

Figure 1.2: Sample flow for two-dimensional function

It is said that f points inwards on a set $\mathcal{B}$ if $x(0) \in \mathcal{B}$ implies $x(t) \in \mathcal{B}\ \forall t \geq 0$. Thus, once the flow enters such a set, it never leaves. The set $\{(x_1, x_2) | x_1, x_2 > 0\}$ constitutes such a set in the above example.

Definition : Inverse of a function

h^{-1} is called the *inverse* of h if $h^{-1}(h(x)) = x\ \forall x \in R^n$

Definition : Homeomorphism

A function $h: R^n \rightarrow R^n$ is a homeomorphism on R^n if and only if:

1. h is one-to-one and onto
2. $h \in C^0$
3. $h^{-1} \in C^0$

Definition : Topological Equivalence of Flows

Two flows g^t and f^t on R^n are topologically equivalent if there exists a homeomorphism $h: R^n \rightarrow R^n$ such that $h(g^t x) = f^t h(x)\ \forall x \in R^n$ and $\forall t \in R$.

1.2.1 z-Transforms, Transfer Functions and Nonlinear Systems

This section will discuss aspects of nonlinear differential equations which do not apply to linear systems. Although there is a very extensive amount of material which falls into this category, only information which is relevant to the motivation and understanding of this thesis is presented here.

z-Transforms and Laplace Transforms - two of the most common tools used by modern engineering analysis - take place within the context of Linear Systems Theory. The switch from analyzing a system in the 'time domain' to 'Laplace domain' is so common that often it is forgotten that such a switch can only be made if the system in question is a linear, time-invariant one. Consider a simple non-linear system defined by the relationship

$$e(k)-.5e^2(k-1)=u(k)+.3u(k-1) \tag{1.16}$$

Note the similarity of this example to the simple linear system

$$e(k)-.5e(k-1)=u(k)+.3u(k-1) \tag{1.17}$$

Equations (1.16) and (1.17) are identical except that the term $.5e(k-1)$ has been changed to $.5e^2(k-1)$. Equation (1.17) may be written in the z-domain as $E(z)=\dfrac{1+.3z^{-1}}{1-.5z^{-1}}U(z)$. However, when the z-Transform is applied to equation (1.16), the quadratic term causes difficulty. Transforming gives

$$\begin{aligned} \mathcal{Z}[\{e^2(k-1)\}] &= \sum_{k=0}^{\infty} e^2(k-1)z^{-k} \\ &= z^{-1}\sum_{k=0}^{\infty} e^2(k)z^{-k} \\ &\neq z^{-1}(E(z))^2 \end{aligned} \tag{1.18}$$

Unlike when equation (1.17) was transformed to the z-domain, it is impossible to factor $E(z)$ out of each term and thereby separate the effects of the input from the effects of the system. In this nonlinear system, the behaviour of the system is intricately linked to the dynamics of the input. Thus in this case, there is no simple resulting expression for a transfer function. Unlike the linear example, it is not possible to say whether or not the nonlinear system will be stable for all stable inputs.

For the system defined by equation (1.17), both the properties of superposition and invariance to scaling apply. However, in (1.16), neither property holds. It is this fact that dictates the applicability of the z-transform. The z-transform relies on superposition to allow specification of the system response as the sum of its response to basis functions.

This example shows that it is impossible to separate the behaviour of a nonlinear system from the input. This is one of the reasons that nonlinear systems are more difficult to analyze than linear ones. There are no simple tests for stability. Furthermore, there is no way to conveniently represent the system as something which is separate from the input. In the next section, it is proven by way of a simple example that the stability for a general nonlinear system, about which nothing is known, cannot be guaranteed without some knowledge of what the input is.

1.2.2 Non Global Stability

This section discusses non global stability, a phenomenon whereby a nonlinear system is stable for some initial values but not for others. The simple example of the previous section is used to illustrate this.

Equation (1.16) illustrates a serious problem associated with nonlinear systems. This simple example exhibits non-global stability. It is simple to verify that the response to an impulse-like sequence, $u(k) = \{a, 0, 0, 0, \ldots\}$ (which is clearly a stable input provided a stays finite) will decay to zero provided that the value of a lies between -2.3736 and 1.7736. But when a is outside of this range, the response grows without bound towards positive ∞. Note that this sort of behaviour cannot occur in a linear system since

a linear system by definition has responses to scaled inputs which are scaled by the same factor, as was stated in property (1.2).

A system which exhibits non-global stability is dangerous from the point of view of system designers. This is because a designer may implement a controller which has been thoroughly tested and appears to be stable, only to discover that vastly different system behaviour can occur if the input lies even just slightly outside the test range.

1.2.3 Input-Dependent Stability

This section discusses a type of nonlinear system which exhibits input-dependent stability. Input-dependent stability is a form of non-global stability, where a system which appears to be stable for a given input can actually become unstable with a slight change to the input.

Consider a nonlinear differential equation of the form

$$\dot{x} = f(x,u) \tag{1.19}$$

If the function $f()$ has a solution x_0 with the property $f(g(t)x_0,u) = g(t)x_0 f_1(u)$, then input-dependent stability can arise. For instance, consider the function

$$f(x,u) = ux \tag{1.20}$$

Assume that some bounded input function u_o exists which produces some bounded output x_o. A stable function u_o is shown by the solid line in Figure 1.3, and its response x_o is the solid line in Figure 1.4.

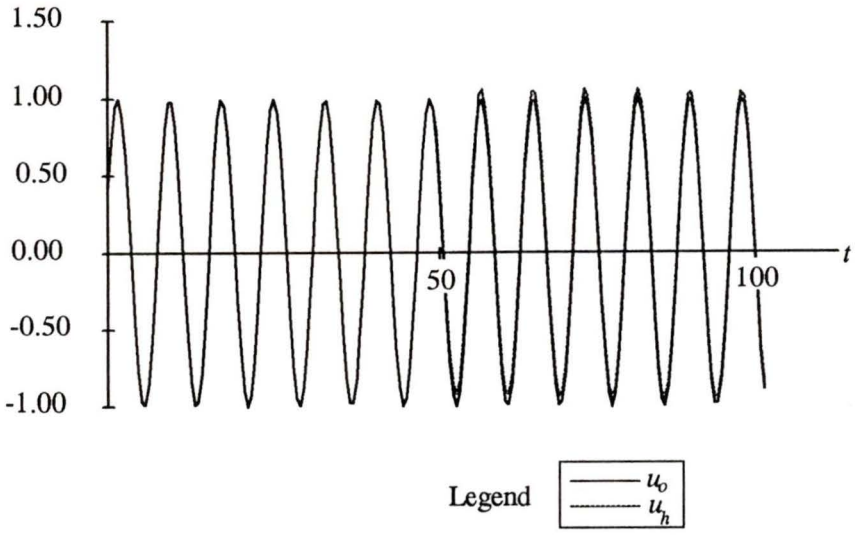

Figure 1.3: Inputs u_o and u_h

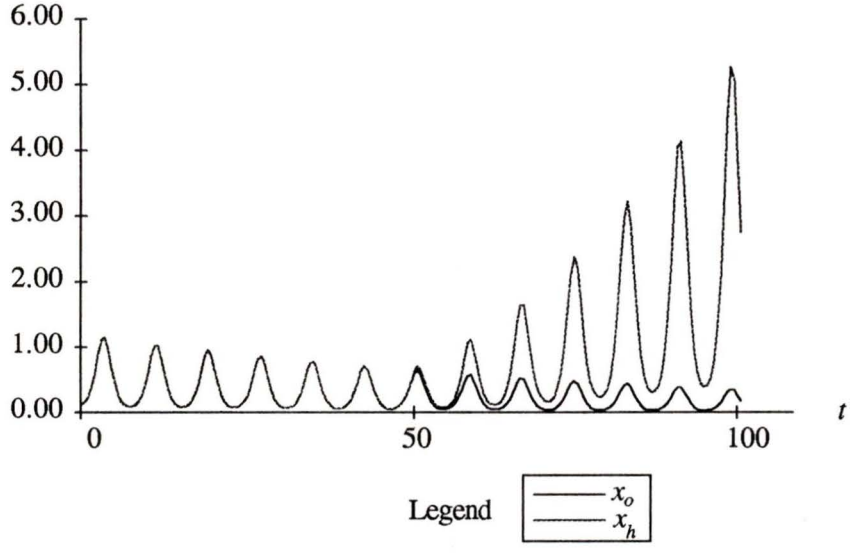

Figure 1.4: Responses x_o and x_h

Now consider a function $x_h = hx_o$, where the function h is to be specified shortly. The question of interest here is, what input is required to give the output x_h? The product rule for differentiation says that

$$\dot{x}_h = \dot{h}x_o + h\dot{x}_o \tag{1.21}$$

and equations (1.19) and (1.20) yield

$$\dot{x}_h = u_h x_h = u_h h x_o \tag{1.22}$$

Equating these two results in a differential equation for h:

$$\dot{h}x_o + h\dot{x}_o = u_h h x_o \tag{1.23}$$

which can be simplified as follows:

$$\begin{aligned} \dot{h}x_o + h\dot{x}_o &= hx_o u_o + hx_o(u_h - u_o) \\ \dot{h}x_o + h(\dot{x}_o - x_o u_o) &= hx_o(u_h - u_o) \\ \dot{h} &= h(u_h - u_o) \end{aligned} \tag{1.24}$$

since $\dot{x}_o = x_o u_o$.

Now let $h = t^k$, where $k>0$. Then $\dot{h} = kt^{k-1}$ and equation (1.24) can be solved to yield

$$u_h = u_o + \frac{k}{t} \tag{1.25}$$

Note that this function approaches u_o as $t \to \infty$. However, since $x_h = hx_o$, the response to this input is $x_h = t^k x_o$, which may grow without bound. Thus two functions which appear to be very similar can in fact produce responses which are topologically distinct.

This is illustrated in Figures 1.3 and 1.4. Figure 1.3 shows two inputs, u_o and u_h which are given by

$$\begin{aligned} u_o(t) &= \sin t \\ u_h(t) &= \begin{cases} \sin t, & t < 50 \\ \sin t + 4/t, & t > 50 \end{cases} \end{aligned} \tag{1.26}$$

The time interval shown is [0...100]. Thus the two inputs are identical for the first half, and after that vary by less than 8%. The responses to these two inputs is shown in Figure 1.4. Clearly, x_h and x_o are identical for the first half, and then x_h begins to grow without bound.

After dealing with linear systems, it is very surprising to see this kind of result. It seems unlikely that such a small difference in the input can produce such a difference in the output. The extra term added to u_h was less than 8% at its maximum, and had decreased to less than 4% by the end of the interval shown here. Terms of this magnitude are often intentionally neglected in the analysis of differential equations.

The reason that this example has been used is to show that nonlinear differential equations can have a response which is highly dependent on the input. It is never possible to state with complete confidence that a nonlinear system is definitely stable since it has been thoroughly tested, since it is never possible to apply all foreseeable inputs to it and measure the response.

The ramifications of this for identification and control are serious. Nonlinear controllers are sometimes used in critical applications, such as the control system of a forward-swept-wing fighter jet[8], where failure of the control system results in catastrophic results. In these situations, the highest priority of a controller must be to remain stable.

In Chapter 2, a traditional design for a nonlinear identification scheme is presented. This design is tested and is known to be a good way of identifying nonlinear systems, but nothing is known about the stability of the resulting model. This is motivation for the original work presented in this thesis, which is the use of neural networks which are known to be stable as a basis for a nonlinear identification model.

1.3 Optimization

This section gives a brief discussion of some well-known optimization techniques. The purpose of this section is to present the techniques of optimization which are used later in this thesis to develop training algorithms for neural networks.

Optimization is concerned with choosing a defined mathematical model which is 'the best' or optimal in some sense. Often, a parameterized model exists for a physical system, in which case the job of optimization is to select the parameters which allow the model to most closely represent the physical system.

Optimization is concerned not only with selecting the best values, but also with the best way to select these values. There is a large number of techniques currently in existence which have been developed to do this[7] , but a discussion of them is beyond the scope of this thesis. This section will primarily discuss two concepts which are used in this thesis. Section 1.3.0 discusses the concept of a cost function, and Section 1.3.1 describes the method of gradient parameter adjustment.

1.3.0 Cost Functions

A cost function takes as inputs the model and the physical process which the model approximates and rates in precise mathematical terms the effectiveness of the model. As an example, consider the linear circuit shown in Figure 1.5. Depending on the initial value of the voltage V_0 on the capacitor, the equation for $V(t)$ is $V(t) = V_0 e^{-t/RC}$. An optimization problem may be to take a model $\tilde{V}(t) = Ae^{-t/\tau}$ and select optimal values for A and τ.

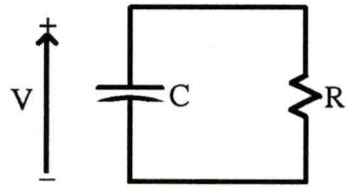

Figure 1.5: Simple Electrical Circuit

One possible measure of the effectiveness of the model is the integral-squared-error:

$$J(\tilde{V}) = \tfrac{1}{2}\int_0^t \left(\tilde{V}(\tau) - V(\tau)\right)^2 d\tau \tag{1.27}$$

Another possible cost function is the instantaneous squared error,

$$J(\tilde{V}) = \tfrac{1}{2}\left(\tilde{V}(t) - V(t)\right)^2 \tag{1.28}$$

The most important criterion for a cost function is that it should have an absolute minimum value. When the cost function has obtained this value, it is then possible to say that the model has attained its best possible result. In the two sample cost functions shown here, this minimum value is 0.

1.3.1 Gradient Methods

This section discusses an optimization technique which is used to adjust the parameters of a model. This technique is called the gradient descent method. The reason for this is that the parameters are adjusted so as to cause the cost function to decrease along the gradient of the cost function with respect to the parameters. This is not always the best way to adjust parameters in the sense that sometimes better adjustments can be made by going in a different direction[7], but it is guaranteed to produce a model which has been improved.

The gradient descent method states that if θ is a parameter of the model, then adjust θ according to the relation

$$\frac{d\theta}{dt} = -\eta \frac{\partial J(\theta)}{\partial \theta} \tag{1.29}$$

The parameter η controls the rate at which the parameter is adjusted. The optimal selection of this value is in itself an optimization problem which is difficult to solve.

Often, a value in the range [.1,.5] is used, although this is merely an empirical rule of thumb[7].

As an example, consider the cost function of equation (1.28). Again consider the simple linear circuit of Figure 1.5, and use as a model $\tilde{V}(t) = Ae^{-t/\tau}$. A gradient parameter adjustment for the parameter A is obtained by using the chain rule for differentiation:

$$\begin{aligned} \frac{dA}{dt} &= -\eta \frac{\partial J}{\partial A} \\ &= -\eta \frac{\partial J}{\partial \tilde{V}} \frac{\partial \tilde{V}}{\partial A} \\ &= -\eta \left(\tilde{V}(t) - V(t)\right) e^{-t/\tau} \end{aligned} \tag{1.30}$$

Using this technique, it is possible to obtain parameter adjustment formulae for all the parameters in the model. This technique will be used later in this thesis to obtain a formula for training a neural network.

1.4 Neural Networks

This section presents an overview of neural networks. Definitions for various kinds of neural networks, including static neural networks and dynamic neural networks, are given. Some of the popular training methods for neural networks are also presented. This section does not cover all facets of neural networks, but rather is intended to introduce the background information which is pertinent to the work in this thesis. For a more thorough overview of neural networks see [19].

The layout of this section is as follows. Section 1.4.0 discusses static neural networks, section 1.4.1 discusses dynamic neural networks, section 1.4.2 presents a method of training called self-organization or Hebbian learning, and section 1.4.3 describes the method of back propagation for training.

1.4.0 Static Nets

As the name implies, static neural networks are governed by a static rather than a dynamic, or differential, equation.

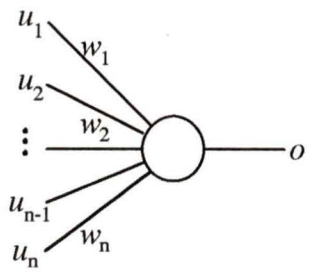

Figure 1.6: A sample neuron

Figure 1.6 shows a typical neuron in a neural network. Each value labeled u_i is a real-valued scalar input to the neuron. The output of the neuron, sometimes called the activation, is labeled o. The values labeled w_i are called the connection weights. The input and output are related by the activation equation:

$$o = f\left(\sum_{i=1}^{n} w_i u_i\right) \tag{1.31}$$

where the function $f()$ can be any desired nonlinear function. Some commonly used functions are the hard-limiter,

$$f_h(x) = \begin{cases} 0, x < \theta \\ 1, x \geq \theta \end{cases} \tag{1.32}$$

the semi-linear function,

$$f_l(x) = \begin{cases} 0, x < 0 \\ x, x \geq 0 \end{cases} \tag{1.33}$$

and the sigmoid nonlinearity:

$$f_s(x) = \frac{1}{1+e^{-\sigma x}} - .5 \tag{1.34}$$

These three functions are illustrated in the following diagrams.

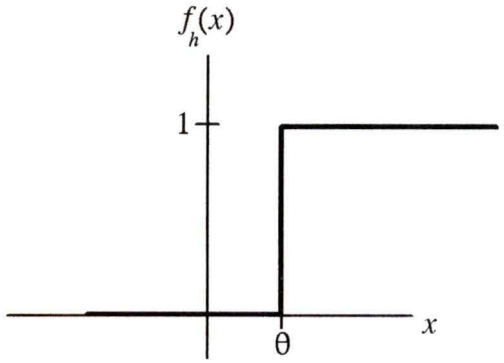

Figure 1.7: Hard limiting neuron activation function

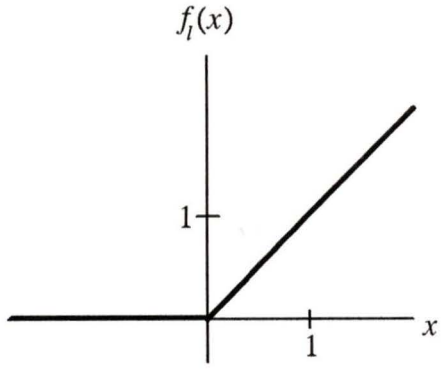

Figure 1.8: Semi-linear neuron activation function

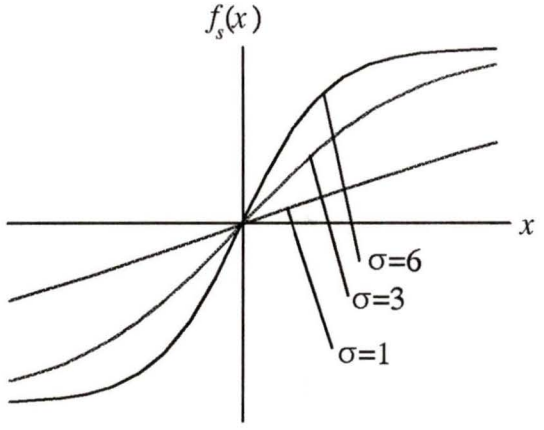

Figure 1.9: Sigmoid neuron activation function

Note that as σ increases for the sigmoid activation function, it begins to resemble an offset version of the hard limiting neuron. Also note that for small values of x the sigmoid function is approximately linear.

When a collection of neurons which all have the same input are arranged together, the resulting grouping is called a *layer*. A layer is shown in Figure 1.10.

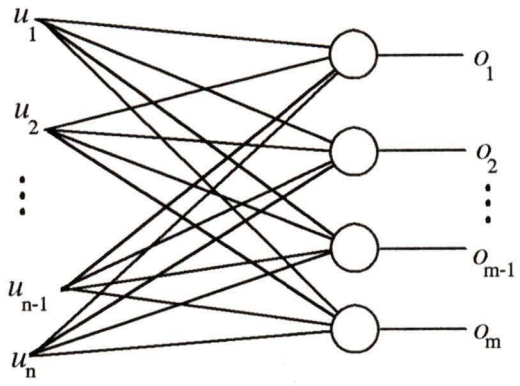

Figure 1.10: A single layer

For convenience, the input and output are sometimes vectorized, and the resulting activation equation is written as

$$\mathbf{o} = \mathbf{f}(\mathbf{W}\mathbf{u}) \tag{1.35}$$

with

$$\mathbf{o} = \begin{bmatrix} o_1 & o_2 & \cdots & o_m \end{bmatrix}^T,$$

$$\mathbf{u} = \begin{bmatrix} u_1 & u_2 & \cdots & u_n \end{bmatrix}^T,$$

$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \\ \vdots \\ \mathbf{w}_m^T \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ & & \vdots & \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{bmatrix} \tag{1.36}$$

This can be represented pictorially as

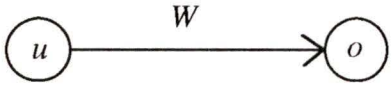

Figure 1.11: Alternate representation for a single layer

When the activation function is some sort of monotonically non-decreasing function, such as the three functions introduced here, a single layer neural network of this variety can be used to implement a least-mean-square-error detector[18]. If there are m hypothesis signal vectors labeled $\mathbf{h}_1$ to $\mathbf{h}_m$, set the weight vectors of the m neurons in the output equal to $\mathbf{h}_1$ to $\mathbf{h}_m$ respectively. When the input is set to $\mathbf{h}_i + \mathbf{n}$, where $\mathbf{n}$ is a noise vector, neuron i will have the largest expected activation.

To see this note that the problem of choosing $\mathbf{w}$ to maximize $\mathbf{h}_i^T\mathbf{w}$ for a given magnitude of vector $\mathbf{w}$ can be solved by using the method of Lagrangian multipliers[7] to change the constrained optimization problem to an unconstrained one. It is therefore necessary to find the solution to the optimization problem

$$J(\mathbf{w},\lambda) = \mathbf{h}_i^T\mathbf{w} + \lambda\left(1 - \mathbf{w}^T\mathbf{w}\right) \tag{1.37}$$

which is easily seen to be given by $\mathbf{w} = \frac{1}{\lambda}\mathbf{h}_i$. Thus, in the neural network described in the preceding paragraph, neuron i will have the largest activation when input $\mathbf{h}_i$ is present due to the non-decreasing activation function. The matter of deciding which signal is present merely involves checking to see which neuron has the largest activation.

Neurons in the single layer configuration shown in Figure 1.10 with a hard-limiting activation function can make decisions about which side of a linear decision boundary the input lies. For instance, in a two-dimensional model, the decision plane as a function of the input is depicted in Figure 1.12.

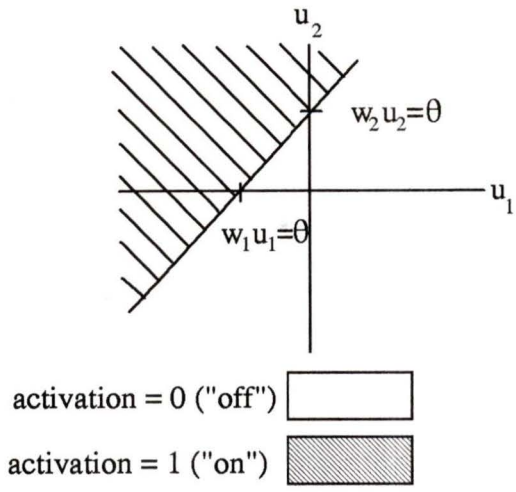

Figure 1.12: Decision boundary for neuron in a single layer

In some applications, this decision boundary may not be complex enough. For instance, a classification problem may require a decision boundary which is curved. In order to implement more complicated boundaries, multilayer neural networks may be used. For instance, a two layer neural network is shown in Figure 1.13.

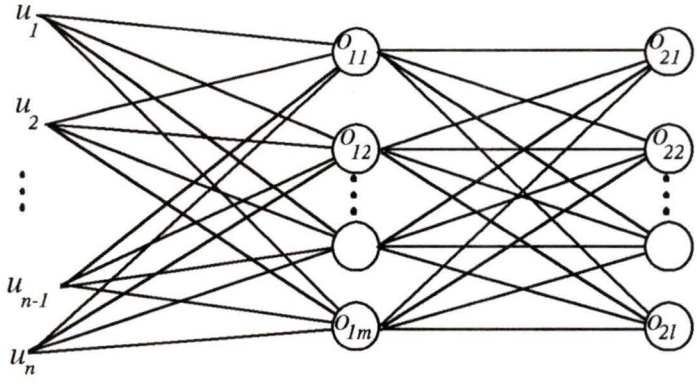

Figure 1.13: A two layer neural network

The two layer neural network can implement curved decision boundaries. However, it cannot implement decision boundaries which have non-contiguous areas which are to be classified together. In order to implement these types of decision boundaries a three-layer

neural network is required. In fact, it can be shown that a three-layer neural network can implement an arbitrarily complex decision boundary provided there are enough neurons in the network [17].

It has been shown [10],[22] that a two layer neural network can approximate any deterministic function over a specified input range with the proper connection weights and with a sufficient number of neurons. In this case, the activation function must be continuous, so the hard-limiter function is not admissible as an activation function. This means that a two layer neural network, which will have a response

$$\mathbf{o} = \mathbf{f}_2(\mathbf{W}_2\mathbf{f}_1(\mathbf{W}_1\mathbf{u})) \tag{1.38}$$

can in theory be made to have the same response as any function,

$$\mathbf{o} = \mathbf{g}(\mathbf{u}) \tag{1.39}$$

1.4.1 Dynamic Neural Networks

The neural networks described in the previous section were called static neural networks since the activation equation contained no dynamics. The neural networks described in this section are called dynamic neural networks since the activation equation is a differential one.

The type of dynamic neural network which is of relevance to this thesis is governed by the activation equation

$$\dot{\mathbf{O}} = -\mathbf{T}\mathbf{O} + \mathbf{W}\mathbf{f}(\mathbf{O}) + \mathbf{b}(t) \tag{1.40}$$

In (1.40), there are N neurons divided into k classes, and

$$\begin{aligned}\mathbf{O} &= \begin{bmatrix}\mathbf{O}_1 & \mathbf{O}_2 & \cdots & \mathbf{O}_k\end{bmatrix}^T \\ &= \begin{bmatrix}o_1 & o_2 & \cdots & o_N\end{bmatrix}^T;\end{aligned} \tag{1.41}$$

is the state of the neural network. The topology of the network is determined by the network connectivity matrix ***W***, which is itself composed of matrices which define how the k classes affect each other:

$$\mathbf{W} = \begin{bmatrix} \mathbf{W}_{11} & \mathbf{W}_{12} & \cdots & \mathbf{W}_{1k} \\ \vdots & & & \\ \mathbf{W}_{k1} & \mathbf{W}_{k2} & \cdots & \mathbf{W}_{kk} \end{bmatrix} \tag{1.42}$$

The $N \times N$ diagonal matrix T,

$$T = \text{diag}\{\tau_1, \quad \ldots \quad, \tau_N\}, \; \tau_i > 0 \tag{1.43}$$

is the matrix of neural relaxation constants. The $N \times 1$ vector $\mathbf{b}(t)$ is the input to the neural network, and $\mathbf{f(O)}$ is some nonlinear function.

A sample dynamic neural network is shown in Figure 1.14 Note the existence of multiple feedback paths. This is the distinction between recurrent neural networks and static feedforward ones as described in section 1.4.0. A feedforward neural network consists of layers, where the output of the first layer is the input to the second, the output of the second is the input to the third, and so forth. A recurrent neural network can have a much more general structure. Dynamic neural networks of this type can also be used to make decisions, much like the static neural networks described in the previous section[5].

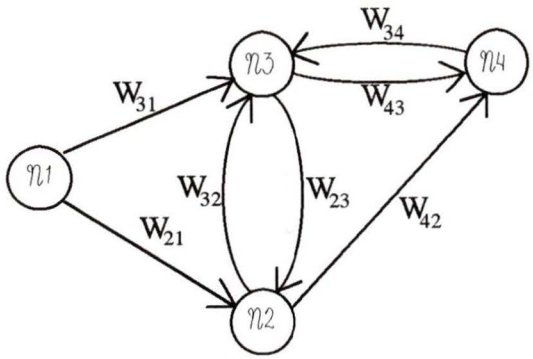

Figure 1.14: Sample Dynamic Neural Network

1.4.2 Back Propagation in Neural Networks

This section describes a technique known as 'the method of back propagation of error' which is commonly used to train the static neural networks described in Section 1.4.0. In the previous sections, it was mentioned that neural networks can be used to make decisions with the proper selection of the connection weights. This section will outline one method used for selecting these weights. Like the gradient optimization methods, this technique adjusts the connection weights in a direction which decreases an error function, which is calculated as a measure of the difference between the actual and correct responses. The chain rule for differentiation is used to obtain formulae which describe the weights' depedence on the error.

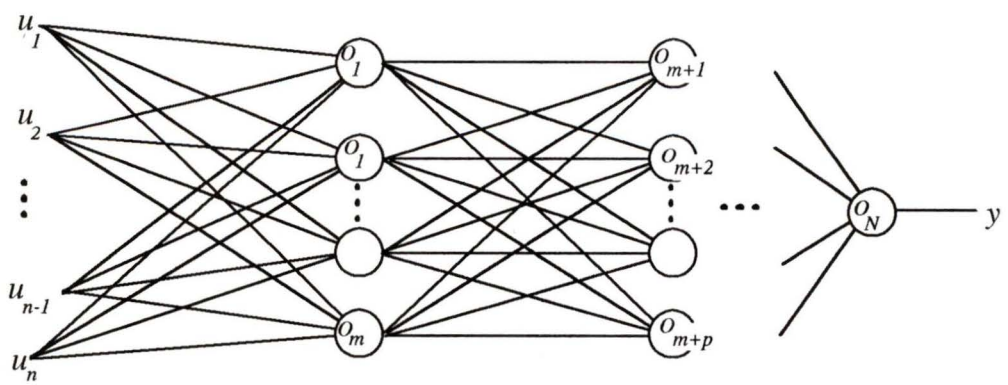

Figure 1.15: Multilayer Neural Network

Figure 1.15 shows an arbitrary multilayer static neural network which has a single neuron in the output layer. The techniques shown in this section can be generalized to neural networks with more neurons in the output layer, but one neuron is used here for clarity.

Assume that the neural network is to be trained so that the output y is to have a specified value in response to several different inputs. The training procedure is as follows. An input pattern is applied to the network, and the output value is calculated by applying equation (1.35) for each layer. An error may then be calculated as

$$e = y - y_d \tag{1.44}$$

where y_d is the desired value of the output neuron.

In a manner analogous to the optimization techniques of section 1.3, define a cost function $J(e)$ as, for instance,

$$J(e) = \tfrac{1}{2} e^2 \tag{1.45}$$

Then, it is desired to adjust the weights of the neural network in a direction opposite to the gradient of the cost with respect to the weights. Thus,

$$\frac{d\theta}{dt} = -\eta \frac{\partial J}{\partial \theta} \tag{1.46}$$

where θ is a connection weight somewhere in the neural network. To compute $\frac{\partial J}{\partial \theta}$, use the chain rule to obtain

$$\frac{\partial J}{\partial \theta} = \frac{\partial J}{\partial o_i} \frac{\partial o_i}{\partial \theta} \tag{1.47}$$

where weight θ is connected to neuron o_i. The second derivative may be calculated from the activation equation:

$$\begin{aligned}\frac{\partial o_i}{\partial \theta} &= \frac{\partial}{\partial \theta} f\left(\sum_j w_{ij} o_j\right) \\ &= f'(s_i) o_j\end{aligned} \tag{1.48}$$

where weight θ is weight w_{ij}, and the notation $s_i = \sum_j w_{ij} o_j$ has been introduced for clarity.

The first derivative in equation 1.47 can also be easily calculated. If o_i is the output node y, then the derivative is merely

$$\frac{\partial J}{\partial o_i} = (y - y_d) = e \tag{1.49}$$

Thus, for weights connected to the output layer, the weight adjustment formula is

$$\begin{aligned}\frac{dw_{ij}}{dt} &= -\eta f'(s_i)(y - y_d) o_j \\ &= -\eta \delta_i o_j\end{aligned} \tag{1.50}$$

where the notation $\delta_i = f'(s_y)(y - y_d)$ has been introduced. For clarity, δ_i will be alternately referred to as δ_y for the output layer.

If o_i is in the layer immediately before the output layer, then note that by the chain rule,

$$\begin{aligned}\frac{\partial J}{\partial o_i} &= \frac{\partial J}{\partial y}\frac{\partial y}{\partial o_i} \\ &= f'(s_y)(y - y_d)w_{yi} \\ &= \delta_y w_{yi}\end{aligned} \tag{1.51}$$

where weight w_{yi} connects o_i to the output. Note that the error term δ_y has been propagated from the output to the preceding layer. This is where the term 'back propagation' is derived. Equation (1.51) and (1.48) may be combined to obtain a weight adjustment formula for neurons in the layer preceding the output:

$$\begin{aligned}\frac{dw_{ij}}{dt} &= -\eta\delta_y w_{yi} f'(s_i)o_j \\ &= -\eta\delta_i o_j\end{aligned} \tag{1.52}$$

where $\delta_i = \delta_y w_{yi} f'(s_i)$.

This procedure may be repeated for each preceding layer, by propagating the error from the layer which has just been adjusted to its input. The general adjustment formula is

$$\begin{aligned}\frac{dw_{ij}}{dt} &= -\eta o_j \sum_k \delta_k w_{ki} \\ &= -\eta\delta_i o_j\end{aligned} \tag{1.53}$$

where the index k runs over all the connections from neuron o_j to the layer above it.

The parameter η controls the rate of adjustment of the weights. If η is too large, then the weight adjustment formula can become unstable, but if it is too small then the training algorithm can take an excessively long time to converge. There is currently no method for determining the optimal value for η.

1.4.3 Self Organization and Hebbian Learning in Neural Networks

The previous section discussed a method for adjusting the neural network weights which was rooted in a mathematical optimization problem. This section discusses a method of training which has its basis in the observation of biological neurons. This is called Hebbian learning. Hebbian learning can be applied to either static or recurrent neural networks. Due to the lack of any other method, Hebbian learning has traditionally been the method of choice for training recurrent neural networks.

In 1949, D.O. Hebb suggested that a type of learning called associative learning occurs in biological neural networks by strengthening the connection of weights between neurons which are both active. In mathematical terms,

$$\frac{dw_{ij}}{dt} \propto o_i o_j \tag{1.54}$$

Note that since the activations are always positive, this implies that the connection weights can only strengthen. In simulated neural networks, it is often useful to add a term which mimics the natural atrophy of neural connections, so that

$$\frac{dw_{ij}}{dt} \propto o_i o_j - k w_{ij} \tag{1.55}$$

where k is some unspecified constant which controls the rate of 'forgetting'.

Note that no mention is made of a reference or desired target value for an output neuron. This is similar to the case in biological neural networks where the network must learn in an unsupervised manner. This type of training can be used to allow the neural network to discover primitives in the input sequence which are relevant to that sequence. For a good example of this type of learning, see [11]

Hebbian learning is not used in this thesis, but it is mentioned here to describe the traditional method of training recurrent neural networks. Chapter 2 discusses a new

method for training such neural networks which, like the back propagation method discussed in the previous section, is based on an optimization problem.

1.5 Chapter Summary

This chapter has provided background information necessary for developing the notions and techniques used in the rest of this thesis. It is beyond the scope of this chapter to give a full treatment of linear system theory, nonlinear differential equations, or neural networks. For a more thorough discussion of linear systems consult [8],[9],[12] or [15]; for nonlinear differential equations consult [3],[9] or [16]; for neural networks see [5],[10],[13],[14] or [17].

Chapter 2: Identification

Introduction

The purpose of this Chapter is to present the theoretical work which has been previously used for identification of unknown nonlinear systems. In developing this work, some linear system theory is presented. The main focus of the discussion on linear systems is to show how linear identification schemes work, and why similar schemes cannot be used for nonlinear systems.

The topic of stability is mentioned throughout this Chapter. It is emphasized frequently that stability is the most important part of any identification procedure which is to be used in any critical application. The Section on linearity also discusses the hazards associated with the stability of nonlinear systems. The reason for this is to motivate the use of neural networks which are proven to be stable as the tool for identification of nonlinear systems.

In this Chapter, linear identification schemes are discussed in Section 2.0. Section 2.1 discusses traditional nonlinear identification schemes, including the use of Volterra Series and neural networks.

2.0 The Linear Model

The main purpose of this Section is to discuss on a theoretical level the reasons underlying the difficulties which nonlinear identification schemes have. Since linear identification methods perform so well, in the first part of this Section, linear methods are discussed in detail, and the reasons that nonlinear systems are more difficult to analyze are shown. This will include a discussion of the method of least squares, and gradient parameter search methods. The applicability of these methods to nonlinear systems is also

covered. Finally, there is a brief discussion of the concept of persistent excitation in linear systems, and how this notion applies to nonlinear systems.

2.0.1 Linear Identification Schemes

This Section uses the z-Transform technique to show that linear identification schemes parameterize the systems they attempt to identify. Then, it is shown why a similar parameterization is impossible for nonlinear systems. In the following Sections there is then a discussion of some well-known linear identification schemes, including the method of least squares, and recursive gradient parameter search methods.

All linear system identification schemes rely on properties 1.1 and 1.2, which were the definitions of a linear transformation. These properties were the properties of invariance under superposition of signals and scaling of signals. This is true because they all assume that the discrete-time system in question may be modeled as

$$\sum_{k=0}^{n} a_k e(t-k) = \sum_{k=0}^{m} b_k u(t-k) \tag{2.1}$$

or for the continuous case,

$$\sum_{k=0}^{n} a_k e^{(k)}(t) = \sum_{k=0}^{m} b_k u^{(k)}(t) \tag{2.2}$$

Thus, the assumption that the system in question is a linear one actually allows a great deal to be known about it. It allows a useful, parameterized model to be assumed for the system. It is a fact that if the order of the system is known, so that n and m in equations 2.1 or 2.2 are known, then these equations will model any and all linear systems, provided the parameters a_k and b_k are identified.

A similarly elegant model is not available for nonlinear systems. The reason for this is that it is not possible to specify a finite set of functions which can determine any and all nonlinear systems. In the nonlinear example of Section 1.2.0, the term $.5e^2(t-1)$ occurred in the difference equation. In order to accommodate this, a parameterized nonlinear model would have to include a term of the sort $a_k e^2(t-1)$ to incorporate the effects of the quadratic term. But this does not allow for the presence of cubic terms, or higher order polynomial terms, trigonometric terms, exponential terms, or other yet-unnamed functions.

Thus any nonlinear identification scheme has a much more difficult job of identification than a linear one since the model it assumes cannot be neatly parameterized into a set of unknowns a_k and b_k. Clearly this is going to have an effect on how quickly or accurately the identification method will converge. This is the main reason that linear identification schemes work so well.

The next two Sections discuss two linear identification methods. These are the method of Least Squares, and Gradient Parameter Search Method.

2.0.2 The Method of Least Squares

A good example of a linear identification scheme is the method of Least Squares. In this method, it is assumed that a linear system exists, and that the system is modeled as:

$$\begin{aligned} y(t+1) &= -a_1 y(t) - \ldots - a_n y(t-n+1) + b_1 u(t) + \ldots + b_m u(t-m+1) \\ &= \varphi^T(t)\theta \end{aligned} \tag{2.3}$$

where

$$\theta = \left[a_1 \cdots a_n \; b_1 \cdots b_m\right]^T$$

$$\varphi(t) = \left[-y(t) \cdots -y(t-n+1) \quad u(t) \cdots u(t-m+1)\right]^T$$

If several samples of the output are considered at once, using the notation

$$Y(t) = \begin{bmatrix} y(1) & y(2) & \cdots & y(t) \end{bmatrix}^T$$
$$\Phi(t) = \begin{bmatrix} \varphi^T(1) \\ \vdots \\ \varphi^T(t) \end{bmatrix} \tag{2.4}$$

then the following equation arises:

$$Y(t) = \Phi(t)\theta \tag{2.5}$$

Gauss was the first to show that the vector θ which minimizes the square error defined as

$$\begin{aligned} V(t) &= \varepsilon^T(t)\varepsilon(t) \\ &= (\Phi(t)\hat{\theta} - Y(t))^T(\Phi(t)\hat{\theta} - Y(t)) \end{aligned} \tag{2.6}$$

is given by:

$$\hat{\theta} = (\Phi^T\Phi)^{-1}\Phi^T Y \tag{2.7}$$

provided that the matrix $\Phi^T\Phi$ is invertible. This can be seen by expanding (2.6) and completing the square :

$$\begin{aligned} V(t) &= Y^T Y - Y^T\Phi\hat{\theta} - \hat{\theta}^T\Phi^T Y + \hat{\theta}^T\Phi^T\Phi\hat{\theta} \\ &= Y^T(I - \Phi(\Phi^T\Phi)^{-1}\Phi^T)Y + \\ &\quad (\hat{\theta} - (\Phi^T\Phi)^{-1}\Phi^T Y)^T\Phi^T\Phi(\hat{\theta} - (\Phi^T\Phi)^{-1}\Phi^T Y) \end{aligned} \tag{2.8}$$

The first term is independent of $\hat{\theta}$, and the second is always non-negative since $\Phi^T\Phi$ is always positive semi-definite. Thus the minimum is obtained by setting the second term to zero, which can be achieved using equation (2.7).

Although this technique can be used to give optimal values for the parameters, it has the drawback that it requires the inversion of a matrix, as well as the calculation of the matrix $\Phi^T\Phi$, which can be potentially time consuming. Furthermore, if this method is used in a recursive manner, so that at each time interval new (and hopefully more accurate) parameters are obtained, a matrix inversion and calculation of $\Phi^T\Phi$ must be performed at each step. This is unacceptably time consuming. In order to combat this, another method which is more suited to a recursive application has been developed. This is the topic of the next Section.

2.0.3 Gradient Parameter Search Method

This technique uses the optimization methods discussed in Section 1.3.1. The gradient parameter search method is a more general technique than the method of least squares. All that it requires is that the system has been parameterized into some set of unknown variables. The technique can be extended to nonlinear systems, as is discussed in detail further in this Chapter.

For linear systems, it is assumed that the model in question has been parameterized as in the preceding Section. Thus the model is defined by

$$\hat{y}(t+1) = \varphi^T(t)\theta \tag{2.9}$$

As above, it is desired to minimize the square error of the difference between the parameterized model and the observed output $y(t)$. Let the instantaneous value for the cost function be

$$\begin{aligned} J(\theta) &= \tfrac{1}{2} e^2 \\ &= \tfrac{1}{2} (\varphi^T(t)\hat{\theta} - y(t))^2 \end{aligned} \tag{2.10}$$

Now consider a recursive parameter adjustment scheme which adjusts the parameters in a direction which causes the error to descend along the gradient:

$$\frac{d\hat{\theta}}{dt} = -\gamma \frac{\partial J}{\partial \hat{\theta}} \tag{2.11}$$

Here, γ is a parameter which controls the rate of adaptation. This derivative can be evaluated by using the chain rule for differentiation:

$$\begin{aligned} \frac{d\hat{\theta}}{dt} &= -\gamma \frac{\partial J}{\partial e} \frac{\partial e}{\partial \hat{\theta}} \\ &= -\gamma e \frac{\partial e}{\partial \hat{\theta}} \\ &= -\gamma e \varphi(t) \end{aligned} \tag{2.12}$$

This is a simple and effective scheme, and if the cost function has a single minimum then this scheme will approach it, provided the system is active enough to allow for identification. This condition is known as the input being persistently exciting.

2.0.4 Persistent Excitation

This Section discusses what is meant by the term persistent excitation, and relates this concept to the invertibility of the matrix $\Phi^T \Phi$ in the method of least squares. Persistent excitation is a concept for linear systems, and its implications and limitations for nonlinear systems is also discussed here.

At the end of Section 2.0.3, it was mentioned that the system in question could be identified provided the input was 'sufficiently active'. A simple example will help to clarify this. Consider the FIR discrete system described by the difference equation

$$x(t) = b_1 u(t) + b_2 u(t-1) + e(t) \tag{2.13}$$

where $e(t)$ is a stochastic signal which could represent the measurement noise. This system has two unknowns. For a first attempt at identification of the unknowns, the step input $u(t) = k$ is applied and the response from time t=1 on is measured. After N measurements, the Grammian of the matrix Φ is easily calculated to be

$$\Phi^T \Phi = N \begin{bmatrix} k^2 & k^2 \\ k^2 & k^2 \end{bmatrix} \tag{2.14}$$

This matrix is clearly singular, and thus parameter identification is not possible. This is to be expected, since equation (2.13) shows that with the step input, the measurement equation is always $x(t) = 2k(b_1 + b_2) + e(t)$. Thus, the two unknowns always appear as a sum, and so can never be uniquely determined.

For a second identification technique, consider applying a single-tone sinusoid of known frequency ω to the system. The Grammian after N measurements (when N becomes large) is

$$\Phi^T \Phi = N \begin{bmatrix} 1 & \cos\omega \\ \cos\omega & 1 \end{bmatrix} \tag{2.15}$$

This will be positive definite, and thus parameter identification when the input is a sinusoid is possible.

When the input results in a Grammian of size nxn which is positive definite, then the input is said to be *persistently exciting of order n*. When this occurs, the input can be

seen from equation (2.7) to be 'sufficiently active' to identify n parameters. If more than n unknowns exist in the model, then the input will be insufficient to allow for identification. In the example of equation (2.13), the step function was incapable of allowing for identification of the two unknowns because a step is persistently exciting of order 1. The sinusoid allowed for identification because a sinusoid is persistently exciting of order 2.

Persistent excitation can be defined as requiring that the Grammian be nonsingular. However, there are other methods to test for persistent excitation which do not actually require the Grammian to be calculated or directly tested[2]. If the system has a finite impulse response, so that the matrix Φ only contains input terms, then it is clear that the Grammian is just the statistical estimate of the autocorrelation matrix of the input. Thus, persistency of excitation can be checked by examining the statistics of the input. Doing so leads to the conclusion that white noise is the 'best' input since samples of it are completely uncorrelated, and so its autocorrelation matrix is always nonsingular, no matter how many samples are used. Thus, white noise is persistently exciting of any order. The opposite of this is a pulse, whose autocorrelation matrix approaches 0 no matter how few parameters are to be identified. A pulse is not persistently exciting of any order.

This discussion of persistent excitation is grounded in the linear model. Thus, its conclusions are not directly applicable to the realm of nonlinear systems. However, due to the lack of anything which is as general for nonlinear systems, the intuitive concepts of persistent excitation may be borrowed. It is expected that identification will be most accurate if white noise is used as the input to the nonlinear system. This, however, is not specific enough. In the simple nonlinear example at the beginning of this Section, it was shown that the dynamics of the system may change drastically for even just small changes in the input. Thus, it is required that the input must cause the output to span the entire desired operating range. This is quite different from a linear system, where white noise of

variance 1 will completely identify a system. Here, if white noise of variance 1 is used to identify the system, and the system is driven with white noise of variance 3, then identification is no longer guaranteed.

2.1 Nonlinear Identification

This Section discusses other efforts at identification of nonlinear systems. The main focus is on work started in 1990 by Narendra et al[14]. This technique, like the one to be discussed in Chapter 3, is based on a neural network model. However, the architecture and dynamics of the network used by Narendra varies significantly from the one used in Chapter 3. The drawbacks of this method are also discussed, as motivation for the original work presented later.

In Section 2.1.1, a well-known nonlinear identification technique known as the method of Volterra Series identification is described. In Section 2.1.2, it is shown how static feedforward neural networks may be connected to form more complex, dynamic systems. Section 2.1.3. provides some information on structural stability, and discusses the implications of this for the dynamic systems of Section 2.1.2. Section 2.1.4 describes a method, called Dynamic Back Propagation, which can be used to train the networks described in 2.1.2.

2.1.1 Volterra Series Identification

This Section gives an overview of a technique first investigated by Volterra early in this century. The approach is an extension of linear analysis techniques which applies to nonlinear systems[3].

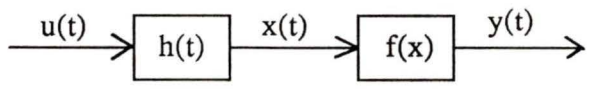

Figure 2.1: Block diagram for Volterra Model

Volterra's approach is to view a nonlinear system as a cascade of a linear dynamic system with a static nonlinearity. Figure 2.1 shows such a system. Many nonlinear systems can be shown to be described by such a model, although a system which has nonlinear feedback, such as a system which exhibits hysteresis, clearly cannot. In Figure 2.1, *h(t)* represents a linear system, which can be described by the relationship

$$x(t) = \int h(\tau)u(t-\tau)d\tau \tag{2.16}$$

Thus, *y(t)* is given by

$$y(t) = f\left(\int h(\tau)u(t-\tau)d\tau\right) \tag{2.17}$$

When the nonlinearity *f(x)* is expanded in a power series, a Volterra series results. The expression is

$$y(t) = \sum_{n=1}^{\infty}\int\cdots\int h_n(\tau_1,\tau_2,\ldots,\tau_n)\prod_{i=1}^{n}u(t-\tau_i)d\tau_i \tag{2.18}$$

The functions $h_i(\tau_1,\cdots,\tau_i)$ are referred to as Volterra kernels. As an example, consider the case when the nonlinearity of Figure 2.1 is given by $f(x) = x + x^2$. Substituting this into equation (2.17) results in

$$\begin{aligned} y(t) &= \int h(\tau)u(t-\tau)d\tau + \left(\int h(\tau)u(t-\tau)d\tau\right)^2 \\ &= \int h(\tau)u(t-\tau)d\tau + \int h(\tau_1)u(t-\tau_1)d\tau_1 \int h(\tau_2)u(t-\tau_2)d\tau_2 \\ &= \int h(\tau)u(t-\tau)d\tau + \int\int h(\tau_1)h(\tau_2)u(t-\tau_1)u(t-\tau_1)d\tau_1 d\tau_2 \\ &= \int h_1(\tau_1)u(t-\tau_1)d\tau_1 + \int\int h_2(\tau_1,\tau_2)u(t-\tau_1)u(t-\tau_1)d\tau_1 d\tau_2 \end{aligned} \tag{2.19}$$

This is seen to be of the same form of equation (2.18).

Using the Volterra series approach reduces the job of identification of the nonlinear system to the identification of the kernels. In practice, this task is quite difficult. First, a convenient method of representing the kernels must be decided upon. The simplest method is a numerical model which stores function values in a large array and uses interpolation to produce approximate values.

Another problem is that although equation (2.19) shows how the input *u(t)* is related to the output *y(t)*, the relationship is very complex, even for this simple second order nonlinearity. If it is possible to take the unknown system offline and drive it with Gaussian white noise, then a method for obtaining $h_1(t)$ and $h_2(t_1,t_2)$ is possible by considering some correlation functions:

$$\begin{aligned}\phi_1(\sigma) &= E[y(t)u(t-\sigma)] \\ &= \int h_1(\tau_1)\mathrm{E}[u(t-\tau_1)u(t-\sigma)]d\tau_1 \\ &+ \int\int h_2(\tau_1,\tau_2)\mathrm{E}[u(t-\tau_1)u(t-\tau_1)u(t-\sigma)]d\tau_1 d\tau_2\end{aligned} \tag{2.20}$$

$$\begin{aligned}\phi_2(\sigma_1,\sigma_2) &= E[y(t)u(t-\sigma_1)u(t-\sigma_2)] \\ &= \int h_1(\tau_1)\mathrm{E}[u(t-\tau_1)u(t-\sigma_1)u(t-\sigma_2)]d\tau_1 \\ &+ \int\int h_2(\tau_1,\tau_2)\mathrm{E}[u(t-\tau_1)u(t-\tau_1)u(t-\sigma_1)u(t-\sigma_2)]d\tau_1 d\tau_2\end{aligned} \tag{2.21}$$

This expression simplifies greatly since the autocorrelation function for white noise is the delta function. Thus,

$$\begin{aligned} \phi_1(\sigma) &= h_1(\sigma) \\ \phi_2(\sigma_1,\sigma_2) &= E[y(t)]\delta(\sigma_1-\sigma_2)+2h_2(\sigma_1,\sigma_2) \end{aligned} \tag{2.22}$$

and the kernels are given explicitly.

Although this technique results in an elegant solution for a second order nonlinearity, the method does not generalize to higher orders. Furthermore, the stipulation that the input must be white noise can be inconvenient in many real systems.

Another series method of nonlinear system identification is the Weiner Series. The primary advantage of the Weiner Series over the Volterra series is that in the former the series is assumed to be made of orthogonal functionals. This simplifies the calculation of the kernels somewhat[3]. Although the results are different from those given above, a full treatment of the Weiner Series is out of the scope of this thesis.

A major difficulty which all series methods have is that they require an extremely large amount of input/output data points in order to ensure persistency of excitation. The number of data points increases as k^n, where k is the number of points required for identification of the first-order kernel, and n is the order of the kernel. This can be clearly seen by observing that the number of parameters of the kernel is equal to its order. k is generally around 30, and so even just a third-order evaluation would require on the order of 27000 data points.

Another disadvantage of this method is that only limited type of systems can be identified using this method. For instance, if nonlinear feedback, such as hysteresis, is involved, then there is no guarantee of the accuracy of the solution[3]. This is because the model assumes the system to be a cascade of a linear dynamic system with a static nonlinearity.

These methods also share a major problem of the feedback networks discussed earlier in Section 2.1. This is the problem of stability. Once again, a complicated model is used which is difficult to analyze. There can be no guarantee that the resulting model will be stable under all the conditions it may face during use. In many applications this concern will preclude the use of this method.

This Section has discussed some of the more common series methods which are available for nonlinear system identification. The difficulties in using these methods has also been discussed.

2.1.2. Feedback Networks

In this Section, a method is described to show how the static feedforward neural networks described in Section 1.4.0 can be used to create a dynamic system. This is achieved through the use of feedback. The implications of this, in terms of stability of the overall system, are also discussed. It should be noted that the work presented in this Chapter and in Section 2.1.4 was originally presented by Narendra et al in [14].

A general, single input, single output (SISO) dynamic system can be approximated by a sampled data, discrete time system as :

$$x(t+1) = f(x(t), x(t-1), \ldots, x(t-m), u(t), u(t-1), \ldots, u(t-n)) \qquad (2.23)$$

This is well-suited for implementation by a neural network. As mentioned in Section 1.4.0, a two-layer neural network, denoted by N, can implement a general function to a specified accuracy. Thus, the configuration shown in the Figure 2.2 can implement the system described by equation (2.23), provided that N can implement $f()$.

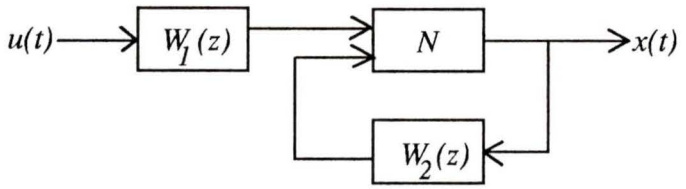

Figure 2.2: A Feedback Neural Network for Implementing Systems Described by (2.19)

Here, $W_1(z)$ and $W_2(z)$ are linear transfer operators which serve to generate the delayed versions of the input and output. Thus the addition of feedback, with the appropriate time-delay units, can create a dynamic system from a static neural network.

Sometimes, it is known that the function *f()* in (2.23) has some specified form. For instance, it may be known that the effects of the input and the state are separable, so that (2.23) becomes

$$x(t+1) = f_1(x(t), x(t-1), \ldots, x(t-m)) + f_2(u(t), u(t-1), \ldots, u(t-n)) \quad (2.24)$$

In this case, a special architecture may be used for the neural network which capitalizes on this information. Using a system composed of two, perhaps smaller, neural networks, such as shown in the following diagram as N_1 and N_2, can perhaps result in a more manageable solution.

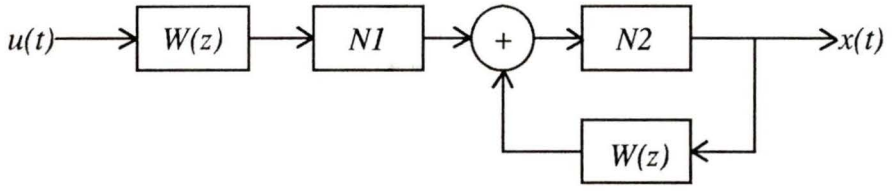

Figure 2.3: A Neural Network to Implement Systems Described by (2.20)

Again, *W(z)* represents a linear transfer function which is used to generate the required delay values. There is intuitive appeal to using this type of a network rather than the more general one in Figure 2.2. In this network, the knowledge that the effect of the input and built-in dynamics can be separated into distinct functions and combined using summation is incorporated by the use of the two neural networks, denoted *N1* and *N2*. When extra information is used, it generally makes sense that the solution will be better. It is true, however, that this type of a neural network is less flexible. For instance, there may be difficulty modeling a system of the type

$$x(t+1) = x(t)u(t) \quad (2.25)$$

since here the feedback is not added to some function of the input, but rather is modulated by it. In this case, the network shown in Figure 2.2 should be used.

This discussion has been included to highlight the fact that matching the attributes of the neural network based model to the known attributes of the system makes sense. In later Sections, this same argument will be used to motivate the use of neurons which have built-in dynamics when modeling dynamic systems.

An extremely important point to note with the system shown in Figures 2.2 and 2.3 is that they may very well be unstable. This is due to the feedback connection. The neural networks involved implement nonlinear functions, and hence it is not a simple matter to check for stability. If they were simple linear systems, it would of course be possible to carry out analysis to show their stability or instability. Furthermore, since there are generally a large number of neurons in the network, the analysis methods which are available for nonlinear systems, such as the use of Liapunov functions or graphical methods, become intractable. Even if the neural network seems to have been trained (such as by using the method described in the next Section) and is following the plant, which is perhaps assumed to be stable, it is still true that since it is a nonlinear system then even the smallest difference between model and plant can mean that one is stable and the other is not. This is the topic of the next Section.

This Section has discussed the basic method for creating a dynamic system from a static neural network. It has also described the chief hazard in this method, which is the problem of stability. The next Section expands more fully on this hazard. A method for training these neural networks for identification follows in Section 2.1.4.

2.1.3 Structural Stability and Implications for Identification

This Section discusses the structural stability of nonlinear systems. The term is first defined and then is explained more fully using some simple examples. This information is used to show a potential hazard of the type of neural network described in Section 2.1.2.

Structural stability is a branch of dynamic system theory which investigates the effects of changing a differential equation on its solutions[9]. A possible definition for structural stability is

Definition: Structural Stability

A vector field $f \in C^1(\mathcal{B})$ which points inwards on the boundary of $\mathcal{B}$ is said to be structurally stable on $\mathcal{B}$ if there exists an $\varepsilon > 0$ such that for all vector fields $\tilde{f} \in C^1(\mathcal{B})$, if $\left\| f - \tilde{f} \right\| < \varepsilon$, then the flow determined by $\tilde{f}$ is topologically equivalent to the flow determined by f.

Simply put, this means that if the differential equation defined by $\dot{x} = f(x)$ is altered slightly, then the DE is said to be structurally stable provided that the solutions before and after the alteration do not vary too drastically.

Of course, structural instability arises when the solution does vary drastically when the equation is modified slightly. A simple example of a system in which this occurs is given by the DE

$$\dot{x} = \mu x - x^3 \tag{2.26}$$

For $\mu \leq 0$, all initial values of x have the same asymptotic solution: $\lim_{t\to\infty} x(t) = 0$. However, if $\mu > 0$, there are stable points of the DE at $x = \pm\sqrt{\mu}$. This is known as a pitchfork bifurcation, a name which results from the shape of the graph which is made when the flow is plotted against μ. This is shown in Figure 2.4.

This sort of behaviour results in limit cycles in two dimensional systems, and even more complicated behaviour for higher dimensional systems. No matter how small μ is made, the behaviour of the systems with $\mu \leq 0$ and $\mu > 0$ will be topologically distinct.

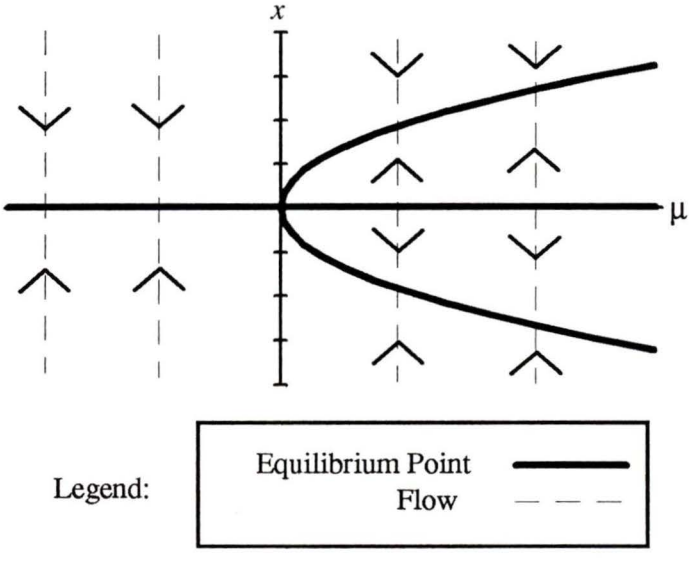

Figure 2.4: Flow vs. μ for equation 2.17

The implication of this for modeling of nonlinear systems is serious. Consider the case when some model is being used to identify a nonlinear system which has some structural instability. For instance, suppose that the system was described by equation (2.26) with $\mu < 0$. As mentioned, this system is globally asymptotically stable. If the model used discovers relation (2.26) but uses a positive value for μ, then a topologically different model results.

In this example, the bifurcation point was easy to see, and it is possible to argue that if a model was on one side of the bifurcation point and the plant on the other, then it would be easy to see this and the model would be changed to rectify the problem. However, many systems contain bifurcation points which are much more difficult to see. Some bifurcation points may have their effects felt for very specific inputs, similar to the cases seen in Chapter 1. If the training procedure for the neural network fails to cover these inputs, then the bifurcation point may not be discovered until after training is done. If the neural net is being used in some critical application, this could have catastrophic results.

It is true that the structural stability of a system will be an inherent property of it, and thus it is arguable that no matter what kind of an identification scheme is being used, the problem will always exist. However, it is also true that given this knowledge it is best to guard against the worst case scenario. Since the type of neural nets shown in Figures 2.2 and 2.3 have no guarantee whatsoever about their stability, they should always be treated, even when they are fully trained, as if they could possibly become unstable under the correct conditions. In Chapter 3, a type of neural network is introduced which can at least guarantee asymptotic stability. This is undeniably preferable to the networks shown here when the network is being used in critical situations.

Given these limitations, it is still useful to investigate more fully what the neural network configurations of Section 2.1.2 can do. In the next Section, a method for training a neural network of the type described in 2.1.2 to follow a given plant is outlined.

2.1.4 Dynamic Back Propagation

This Section discusses the extension of the backpropagation methods introduced in Chapter 1 to the dynamic configurations of neural networks introduced in the previous Section. This is a technique which has been called dynamic back propagation by Narendra et al[14]. Some results of this technique when applied to simple systems are also shown.

In the latter part of this Section, a simplification to the method is made which allows static backpropagation to be used. This simplifies the training procedure. The effects of this simplification are also discussed.

Consider the following system, which is similar to those discussed in the previous Section:

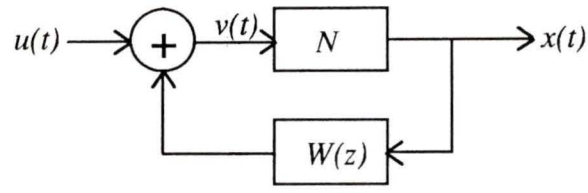

Figure 2.5: Architecture for backpropagation

The task of developing a backpropagation method for this system is quite similar to developing one for a static neural network. As in Chapter 1, the chain rule for derivatives is used extensively. Define the cost function as:

$$\begin{aligned} J &= J(e) \\ &= J(x(t) - \hat{x}(t)) \end{aligned} \tag{2.27}$$

As in Chapter 1, the rule used to minimize this is a gradient descent rule. If θ is a parameter, i.e., a connecting weight, within the neural network, then the gradient descent rule says to use the adjustment formula:

$$\begin{aligned} \frac{d\theta}{dt} &= -\gamma \frac{\partial J}{\partial \theta} \\ &= -\gamma \frac{\partial J}{\partial x} \frac{\partial x}{\partial \theta} \end{aligned} \tag{2.28}$$

It is interesting to note the similarity of this method to that of the gradient parameter search method discussed in Section 2.0.3. Equation (2.28) shows that what is required to be known is the derivative of the output with respect to each parameter which is to be

adjusted. Unlike the static networks discussed in Chapter 1, the derivative $\partial x / \partial \theta$ is actually obtained as the solution to a differential equation. For clarity, let $\partial x / \partial \theta \equiv d(t)$. From Figure 2.4, it is seen that

$$d(t) = \frac{\partial N(v)}{\partial \theta} \tag{2.29}$$

Using partial differentiation, and keeping in mind that v is also a function of θ , this can be rewritten as

$$d(t) = \frac{\partial N}{\partial v}\frac{dv}{d\theta} + \frac{\partial N}{\partial \theta} \tag{2.30}$$

This may be further rewritten to show the dynamics of this equation:

$$\begin{aligned} d(t) &= \frac{\partial N}{\partial v}\frac{dv}{d\theta} + \frac{\partial N}{\partial \theta} \\ &= \frac{\partial N}{\partial v}\frac{d(u + W(z)x)}{d\theta} + \frac{\partial N}{\partial \theta} \\ &= \frac{\partial N}{\partial v}W(z)\frac{dx}{d\theta} + \frac{\partial N}{\partial \theta} \\ &= \frac{\partial N}{\partial v}W(z)d(t) + \frac{\partial N}{\partial \theta} \end{aligned} \tag{2.31}$$

Due to the presence of the transfer operator $W(z)$, $d(t)$ is calculated as the output of a linear difference equation. The partial derivatives $\partial N / \partial v$ and $\partial N / \partial \theta$ are easy to calculate using the static backpropagation methods discussed in Chapter 1. If it is

assumed that the parameter γ in equation (2.28) is small, so that θ can be assumed to be constant compared to the variations of x, then it can be said that using (2.31) to adjust θ will result in a stable parameter adjustment rule, since $W(z)$ is certainly stable. This is to say that the parameters themselves will remain bounded; however, there is no guarantee on the stability of the system which *uses* these parameters.

Equation (2.31) provides the means to assure a gradient adjustment of the parameters, but it is rather complicated. It shows that for each parameter in the network, a certain number of derivatives, equal to the order of W(z), will have to be calculated at each point that a parameter adjustment is desired. This is time consuming since there are usually a large number of parameters, i.e., 20 for a small neural network, 500 or more for a large one.

To help alleviate this problem, Narendra[14] has proposed a simplification to the calculation of the gradient $\partial x / \partial \theta$. In the simplification, the dynamics of the model which arise from the feedback connection are ignored. Instead, it is assumed that the error at the current instant of time is a function of the current state of the network and input only, and the past states of the model do not affect the current error. For such a neural network, static backpropagation can be used. For the purpose of the correction of the weights, the model shown in Figure 2.5 effectively becomes the one shown in Figure 2.6. The error is propagated from the output to the input of the neural network in the manner described in Section 1.0.4.

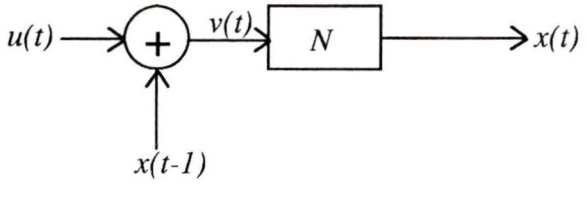

Figure 2.6: Effective Simplified Architecture for Back Propagation

2.1.5 Summary of Feedback Networks

In Sections 2.1.0 to 2.1.4, it was shown that it is possible to design a model of a dynamic system using static neural networks by connecting a feedback loop. It was mentioned that such a network could implement an arbitrary system provided the order of the system was known and there were sufficient neurons in the network.

Section 2.1.3 discussed the largest problem with this method, which is that the stability of the neural network model is never assured. This is a problem when dealing with nonlinear systems since structural instability can sometimes arise. This can lead to a model which seems to be giving good following but is actually unstable.

Section 2.1.4 developed a good method for training the dynamic neural network models of the earlier Sections. In addition, a simplification was made which allowed the less complicated technique of static back propagation to be applied to the training of dynamic models.

Chapter 3: Stable Neural Networks for Identification of Nonlinear Systems

Introduction

This Chapter focuses on a class of dynamic recurrent neural networks which are known to be stable. A training method, including a logical motivation for the choice of an architecture, as well as a procedure for training the neural network, is presented. Although the stability of the neural networks has been previously shown[5], the application of these neural networks to the complex task of system identification is novel.

The organization of this Chapter is as follows. Section 3.0 gives an overview of the class of stable neural networks. Section 3.1 devlops a training procedure for this class, and Section 3.2 devolps a logical choice for an architecture for using such neural networks for identification of nonlinear systems.

3.0 Stable Neural Networks

This Section gives a summary of a class of stable neural networks. This class of neural networks is discussed in full detail in [5]. This Section is included to give an overview of the work of [5]. There, it is shown that asymptotic stability is ensured for neural networks which are described by the differential equation

$$\dot{\mathbf{O}} = -\mathbf{T}\mathbf{O} + \mathbf{W}\mathbf{f}(\mathbf{O}) + \mathbf{b}(t) \tag{3.1}$$

In (3.1), there are N neurons divided into k classes, and

$$\begin{aligned} \mathbf{O} &= \begin{bmatrix} \mathbf{O}_1 & \mathbf{O}_2 & \cdots & \mathbf{O}_k \end{bmatrix}^T \\ &= \begin{bmatrix} o_1 & o_2 & \cdots & o_N \end{bmatrix}^T; \end{aligned} \tag{3.2}$$

is the state of the neural network. The topology of the network is determined by the network connectivity matrix W:

$$\mathbf{W} = \begin{bmatrix} \mathbf{W}_{11} & \mathbf{W}_{12} & \cdots & \mathbf{W}_{1k} \\ \vdots & & & \\ \mathbf{W}_{k1} & \mathbf{W}_{k2} & \cdots & \mathbf{W}_{kk} \end{bmatrix} \tag{3.3}$$

The $N \times N$ diagonal matrix T,

$$\mathbf{T} = \text{diag}\{\tau_1, \quad \ldots \quad, \tau_N\} \tag{3.4}$$

is the matrix of neural relaxation constants. The $N \times 1$ vector $\mathbf{b}(t)$ is the input to the neural network, and $\mathbf{f(O)}$ belongs to the class of so-called neuromime functions, which are defined as $\aleph = \{\mathbf{f} | \mathbf{f}: R^N \rightarrow R_+^M, \mathbf{f}$ continuous, $\mathbf{f}$ monotonically non-decreasing, satisfying a Lipschitz condition and $\exists \theta \in R^N$ such that $\mathbf{f}(\theta) = \mathbf{0}\}$.

In [5], it is shown that the sufficient condition on $\mathbf{W}$ that guarantees asymptotic behaviour is that it must contain all of its positive entries on one side of the main diagonal. This gives an easy way to check whether a neural network is stable. For instance, the neural network shown in Figure 3.1 has a connectivity matrix given by

$$\mathbf{W} = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{W}_{21} & \mathbf{0} & \mathbf{W}_{23} & \mathbf{0} \\ \mathbf{W}_{31} & \mathbf{W}_{32} & \mathbf{0} & \mathbf{W}_{34} \\ \mathbf{0} & \mathbf{W}_{42} & \mathbf{W}_{43} & \mathbf{0} \end{bmatrix}. \tag{3.5}$$

It was shown [5] that the neural network governed by the differential equation (3.1) is stable in the sense described below provided that the connection weights in submatrices $\mathbf{W}_{23}$ and $\mathbf{W}_{34}$ are non-positive (i.e., inhibitory). Alternately, stability is ensured if the weights in submatrices $\mathbf{W}_{21}$, $\mathbf{W}_{31}$, $\mathbf{W}_{32}$, $\mathbf{W}_{42}$ and $\mathbf{W}_{43}$ are non-positive.

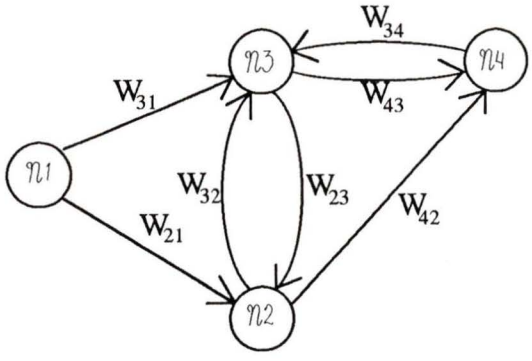

Figure 3.1: Sample Neural Network

The term stability is applied to the neural network shown in Figure 3.1 as follows. Essentially, when the input to the neural network is held constant, the state **O** evolves and is bounded above and below by a so-called poly-exponential function. Precisely, the system

$$\dot{\mathbf{O}} = -\mathbf{TO} + \mathbf{Wf}(\mathbf{O}) + \mathbf{b} \tag{3.6}$$

where **b** is an $N \times 1$ constant vector, has solutions $\mathbf{O}(t, \mathbf{O_0})$ which are bounded as

$$\mathbf{K} - e^{-\mathbf{A}t}\mathbf{P}(t) \le \mathbf{O}(t, \mathbf{O_0}) \le \hat{\mathbf{K}} - e^{-\hat{\mathbf{A}}t}\hat{\mathbf{P}}(t) \tag{3.7}$$

where $\mathbf{A}$ and $\hat{\mathbf{A}}$ are $N \times N$ positive diagonal matrices, $\mathbf{P}(t)$ and $\hat{\mathbf{P}}(t)$ are $N \times 1$ vector polynomials of finite order, and $\mathbf{K}$ and $\hat{\mathbf{K}}$ are $N \times 1$ constant vectors. The less-than operator is applied to $N \times 1$ vectors meaning

$$\mathbf{K_1} \le \mathbf{K_2} \Leftrightarrow K_{1_i} \le K_{2_i} \ \forall i = 1, 2, \ldots, N. \tag{3.8}$$

One important point to note is that the class of neuromime functions maps into the positive reals. Thus, any neural network defined by (3.1) will have a positive output. This may pose a problem for identification and control applications, which generally require both positive and negative values. However, this problem is not serious since in

general a neural network, or any nonlinear identification scheme, will be computer-based, and therefore will be dealing with a sampled data stream. Thus, it is possible to calibrate the digital-to-analog and analog-to-digital converters so that the input and output for the neural network ranges from 0 to 2^n, or a scale factor may be introduced so that the input and output vary between 0 and 1. Thus, the positive-only restriction is not a problem.

This Section has presented a class of neural networks which are globally uniformly bounded by poly-exponential functions. The test for this stability was seen to be very simple to use in an arbitrarily complex neural network. These neural networks have been trained in the past to perform various non-dynamic tasks, such as performing an exclusive-or of two inputs[5]. The neural networks have been trained using Hebbian learning.

3.1 Training procedure for Stable Neural Networks

This Section discusses a method for adjusting the weights and other parameters of neural networks that are stable in the sense described in Section 3.0. The general approach that is used here is to define a criterion and then adjust the parameters in a direction that will decrease this cost. In this sense the technique is similar to linear recursive adaptive methods [2] and to classical back propagation [17]. However, since the stable neural networks described in Section 2.2 have certain restrictions on the polarity of the connection of classes, a straightforward gradient adjustment is not possible. A solution for this is also presented here.

3.1.1 Gradient of Cost Function

The general equation for calculating the behaviour of the class of neural networks of interest here is

$$\dot{\mathbf{O}} = -\mathbf{TO} + \mathbf{Wf(O)} + \mathbf{b} \tag{3.9}$$

using the same notation introduced in Section 3.0. One possible criterion for measuring the performance is the quadratic cost function

$$\begin{aligned} \mathbf{J(e)} &= \mathbf{1/2(O-O_d)^T A(O-O_d)} \\ &= \mathbf{1/2\, e^T Ae} \end{aligned} \tag{3.10}$$

where $\mathbf{O_d}$ is the desired state of the neural network. Matrix **A** is used to eliminate from the cost any neurons whose state is not crucial. **A** is a diagonal matrix with *1*'s corresponding to output neurons and *0*'s elsewhere. As in other recursive adaptive methods[2], parameters θ in the neural network are adjusted along the negative gradient of this cost, i.e.,

$$\frac{d\theta}{dt} = -\eta \frac{\partial J}{\partial \theta} \tag{3.11}$$

The chain rule for differentiation is used to allow for the calculation of this gradient for parameters associated with neuron *j*:

$$\begin{aligned} \frac{\partial J}{\partial \theta} &= \frac{\partial J}{\partial o_j} \frac{\partial o_j}{\partial \theta} \\ &= \gamma_j \frac{\partial o_j}{\partial \theta} \end{aligned} \tag{3.12}$$

The notation γ_j is used to denote the derivative of the cost with respect to the activation of neuron *j*. If neuron *j* is an output neuron, this derivative is simply given by

$$\gamma_j = o_j - o_{d_j} \tag{3.13}$$

In a manner analogous to traditional back propagation of the error [19], this gradient may be calculated for units that are not output neurons by using the values of the gradient in all the neurons k that have neuron j as inputs:

$$\begin{aligned}\gamma_j &= \sum_k \gamma_k \frac{\partial o_k}{\partial o_j} \\ &= \sum_k \gamma_k \Delta_{kj}\end{aligned} \tag{3.14}.$$

Here, the notation Δ_{kj} has been introduced to represent the partial derivative $\partial o_k / \partial o_j$. To calculate Δ_{kj}, it is necessary to use the differential equation which defines the behaviour of the neural network. Rewriting equation (3.9) specifically for neuron k, and using the operator D to represent differentiation results in

$$(\tau_k + D)o_k = \sum_j w_{kj} f(o_j) + b_k \tag{3.15}$$

Differentiating (3.15) with respect to o_j results in

$$\dot{\Delta}_{kj} = -\tau_k \Delta_{kj} + w_{kj} f'(o_j) \tag{3.16}$$

Note that unlike classical back propagation, the equation governing the propagation of error from one class to the next is a differential equation.

All the derivatives that are required in equation (3.12) to adjust a parameter θ have now been obtained, except for the derivative $\partial o_j / \partial \theta$. The next Section discusses the case when θ is a connecting weight, and the Section following that discusses the case of parameters of the differential equation, such as the relaxation constant τ or parameters of the activation function $f()$.

3.1.2 Input Weight Adjustment

Let θ represent a connecting weight w_{ji} which connects neuron i (input) to neuron j. Use the notation $\xi_{ji} = \partial o_j / \partial w_{ji}$. Using equation (3.15), and differentiating with respect to w_{ji}, the differential equation for ξ_{ji} is obtained :

$$\dot{\xi}_{ji} = -\tau_j \xi_{ji} + f(o_i) \tag{3.17}$$

Again, unlike classical back propagation, this equation is a differential equation. Using this equation and the results of the previous Section, equation (3.12) may now be written as

$$\frac{dw_{ji}}{dt} = -\eta \gamma_j \xi_{ji} \tag{3.18}$$

with γ_j calculated using equation (3.13) or (3.14) as appropriate.

3.1.3 Adjustment of Structural Parameters

The same analysis that was used to determine how to adjust the connection weights can also be used to obtain a formula for adjusting any of the other variables that parameterize the neural network. For instance, a formula for adjusting a parameter of the activation function $f()$ or the neural relaxation constants τ_j (analogous to time constants for a linear system) in a way that will reduce the cost function can be obtained . Consider an activation function of the form

$$f(o) = \begin{cases} \dfrac{1}{1+e^{-\sigma o}} - .5, o \geq 0 \\ 0, o < 0 \end{cases} \tag{3.19}$$

The parameter σ controls how much input causes the neuron to saturate; the larger it is, the harsher the nonlinearity. Since there is no way of knowing *a priori* what an optimal or even appropriate value for this variable is, it makes sense to adapt it while training the connection weights. Since the value γ_j will already be available during training of the weights, the only further calculation required is the derivative $\nu_j = \partial o_j / \partial \sigma_j$. Using equation (3.15), it is seen that

$$\begin{aligned}(\tau_j + D)\nu_j &= \sum_i w_{ji} \frac{\partial f(o_i)}{\partial \sigma_j} \\ &= \begin{cases} \sum_i w_{ji} o_i (.25 - f(o_i)^2), o_i \geq 0 \\ 0, o_i < 0 \end{cases}\end{aligned} \tag{3.20}$$

Using this formula to calculate the value for ν_j, the nonlinearity parameter may be adjusted using the relation

$$\frac{d\sigma_j}{dt} = -\eta \gamma_j \nu_j \tag{3.21}$$

Using the same technique, it is possible to calculate $\beta_j = \partial o_j / \partial \tau_j$ to adjust the relaxation constants. This is useful since it is not know beforehand how fast a system the neural network will be trying to identify. The technique from above yields the differential equation for β_j :

$$\ddot{\beta}_j + 2\tau_j \dot{\beta}_j + \tau_j^2 \beta_j = -\sum_i w_{ji} f(o_i) \tag{3.22}$$

and an update formula for τ_j of

$$\frac{d\tau_j}{dt} = -\eta\gamma_j\beta_j \tag{3.23}$$

It is important here that the relaxation constant τ_j is not adjusted at too great a rate, or else equation (3.22) is not valid since τ_j is a treated as a constant.

3.1.4 Weight Clamping

Section 3.0 describes a class of neural networks that are asymptotically stable. This condition is guaranteed provided that the connectivity matrix W has all of its positive entries on one side of the diagonal [5]. However, equation (3.18) gives a formula for adjusting the connection weights that may violate this condition. To combat this, it is necessary to check the polarity of certain crucial weights after each weight adjustment. For instance, as discussed in Section 3.0, if the weights labeled W_{23} in Figure 3.1 are guaranteed to be non-positive, then the neural network will be stable. Thus after any weight in W_{23} is adjusted using (3.18), the weight should be checked to ensure that it is not positive. If it is, then it should be clamped at 0. This technique ensures that inhibitory weights stay inhibitory throughout the training procedure.

3.2 Stable Neural Network for Identification

This Section uses the results of the previous Section to motivate a neural network architecture suitable for identification of nonlinear systems. In Section 2.1, it was mentioned that using knowledge about the system to be identified can result in a more efficient model. Similarly, this Section discusses some general properties of nonlinear systems and uses these properties to motivate a suitable identification architecture. Specifically, a nonlinear system is described as being a linear system whose 'poles' move

depending on the level of the output or input. A type of neural net, called a scheduler class, is presented which takes advantage of this kind of a nonlinearity.

3.2.1 Nonlinear Systems

This Section discusses nonlinear systems as linear systems which have 'poles' that move depending on the level of the input or output. Consider the simple nonlinear system described by the relation

$$\dot{y} = u - \frac{y}{1+4y^2} \tag{3.24}$$

If y remains relatively constant near some value y_{ss}, then this system can be approximated by a first order linear system that has a pole at $-(1+4y_{ss}^2)^{-1}$. This kind of approximation is often used to design a linear controller for a nonlinear system which is at some operating point[8][9]. If y varies from this value significantly, then the 'pole' can be thought of as roving in some sense. Although strictly speaking this is not an exact description of the behaviour of the system, it does illustrate one of the more common types of nonlinearity which is encountered in real systems such as valve flows and airplanes cruising at various velocities.

A model can be designed to take advantage of this kind of nonlinearity. A linear model can approximate the nonlinear system over a small range of operation, and many such linear models can be designed to approximate the entire system at different points. Then, a scheduler can be used to determine which particular linear model is used, based on the level of the inputs and outputs of the overall system. This technique is often used to design nonlinear controllers using linear control theory[8]. One of the major difficulties is producing a smooth transition when the scheduler determines it is time to switch from one simple model to another. Another problem is developing the linear models, which can be

numerous if there are more than one input or output, or if the system varies significantly from range to range.

In the following Sections, a neural network model is proposed which will do the scheduling and modeling described here. This is a new technique for neural network modeling since it uses a logical choice for the design of the architecture rather than using an all-purpose neural network [14]. As in Section 2.1, the purpose of using knowledge about the way a system operates is used to produce a model which should be more efficient.

3.2.2 A Simple Neural Network for Identification

This Section describes the architecture for a simple neural network which is shown to be able to implement general first and second order linear systems. Since the method of partial fraction expansion can be used to break a linear system of any order into the parallel combination of first and second order responses, it follows that using enough of these simple neural networks in parallel will result in a model capable of identifying an arbitrary linear system.

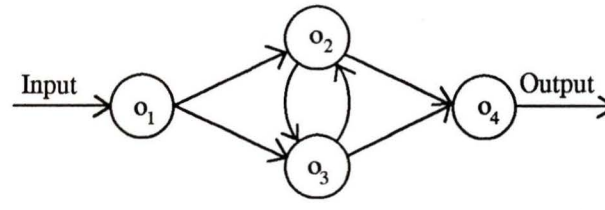

Figure 3.2: A simple neural network for Identification

Figure 3.2 shows a neural network containing 4 neurons whose dynamic behaviour is described by equation (3.1). The results of Section 3.0 state that this neural network will be stable provided that one (or both) of the feedback weights connecting o_2 to o_3 is inhibitory. Assume for convenience of notation that o_3 inhibits o_2. In this Section, the activation function takes the form

$$f(o)=\begin{cases}0, o<0\\ o, o\geq 0\end{cases} \tag{3.25}$$

Furthermore, it is assumed that the input is biased so that the state of the neural network always stays positive. Thus, the activation function is essentially linear and may be ignored in equation (3.1).

The evolution equation for o_4 is then seen to be

$$\dot{o}_4 = -\tau_4 o_4 + w_{42} o_2 + w_{43} o_3 \tag{3.26}$$

If the relaxation constant τ_4 is small compared to the time constants of the other neurons, then o_4 merely tracks its inputs. Thus, the input/output relation takes on the familiar form of a state space equation:

$$\begin{bmatrix}\dot{o}_2\\ \dot{o}_3\end{bmatrix}=\begin{bmatrix}-\tau_2 & -w_{23}\\ w_{32} & -\tau_3\end{bmatrix}\begin{bmatrix}o_2\\ o_3\end{bmatrix}+\begin{bmatrix}w_{21}\\ w_{31}\end{bmatrix}o_1$$
$$o_4=\tau_4^{-1}\begin{bmatrix}w_{42} & w_{43}\end{bmatrix}\begin{bmatrix}o_2\\ o_3\end{bmatrix} \tag{3.27}$$

where w_{23} is positive to ensure the stability of the neural network. Using eigenvector analysis, the poles of this system will be located at the roots of the equation

$$\det\left(\lambda \mathbf{I}-\begin{bmatrix}-\tau_2 & -w_{23}\\ w_{32} & -\tau_3\end{bmatrix}\right)=0 \tag{3.28}$$

which are given by

$$\lambda_1 = -\tfrac{1}{2}(\tau_2 + \tau_3) + \frac{\sqrt{(\tau_2 + \tau_3) - 4(\tau_2\tau_3 + w_{23}w_{32})}}{2}$$
$$\lambda_2 = -\tfrac{1}{2}(\tau_2 + \tau_3) - \frac{\sqrt{(\tau_2 + \tau_3) - 4(\tau_2\tau_3 + w_{23}w_{32})}}{2} \qquad (3.29)$$

The only restriction here, imposed to ensure stability of the neural network, is that w_{23} is positive. If w_{23} is fixed at some value, then the system of equations in (3.29) is still two equations in three variables, and so has an infinite number of solutions for any desired value of λ_1 and λ_2. Thus, the poles of the system may be placed arbitrarily.

As mentioned at the beginning of this Section, a linear system of higher order can be described by a parallel combination of second order responses. Thus, if more neurons are added in classes 2 and 3, the neural network shown in Figure 3.2 can implement a linear system of any order.

In general, the activation function used will not be the linear one shown in equation (2.59). However, if the function is continuous, then the results of this system will still apply provided that the neurons operate within their linear range.

3.2.3 Scheduler Neurons

This Section discusses a neural network which is shown to be capable of implementing the scheduling task as required in the overall nonlinear system as discussed in the introduction to this Section. This type of neural network is called a scheduler class.

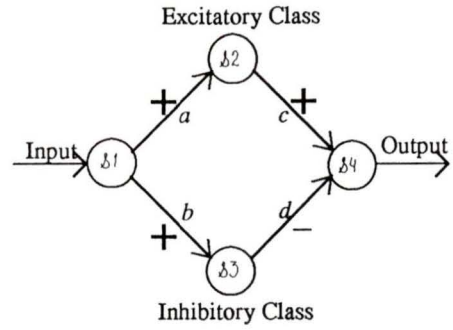

Figure 3.3 : Architecture for Scheduler Network

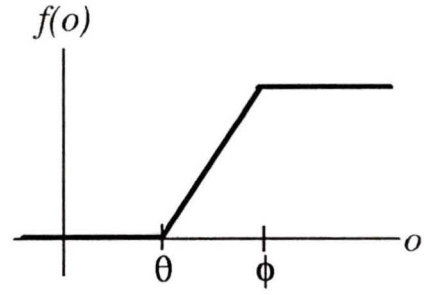

Figure 3.4: Activation Function for Neurons in the Scheduler Network

Figure 3.3 shows a simple neural network whose output can have a peaked response with proper selection of the weights *a,b,c* and *d*. A peaked response means that the neuron will have an activation of zero for low-level inputs, a positive activation for medium level inputs, and again turn off for high-level inputs. The weights of the neural network have the connection polarity marked on the diagram, and stability is ensured since there is no feedback loop. The activation function used has a response as shown in Figure 3.4. The theory of operation now follows.

When the input is small, the fact that the threshold $\theta > 0$ ensures that there will be no output. When the activation of the input class increases to the level θ, both the excitatory class and the inhibitory class will begin to have a non-zero output.

Let $a > b$. Then the output of the excitatory class will always be larger than the output of the inhibitory class. This can be seen by examining the steady state value of a neuron which follows equation (3.1). This steady state value is seen by setting the derivative to zero, which is a valid way of determining this steady state value since it is known that the network is asymptotically stable. Assuming that all neurons have the same time constant, the steady-state output of the excitatory class is

$$o_2 = \tfrac{1}{\tau} af(o_1)$$

while that of the inhibitory class is

$$o_3 = \tfrac{1}{\tau} bf(o_1)$$

Clearly the excitatory class will have a larger output since $a>b$. Now consider the output class. Let the connection weight d be larger in magnitude than the weight c, although as mentioned d will be inhibitory and c will be excitatory. Since the output of the excitatory class is larger than that of the inhibitory class, there will be a region where the excitatory class has passed the threshold θ of the output class but the output of the inhibitory class has not. During this region, the output class will turn on.

When the input becomes very large, both the output of the inhibitory class and the excitatory class will have passed the threshold ϕ, i.e., they will be in the saturation region. Since the magnitude of d is larger than that of c, the net effect will be that the output class will be inhibited. This will turn the output off.

Thus, the overall response of the output class is peaked. It will be off for low inputs, on for medium level inputs, and off for high inputs. This sort of response can be used for scheduling. By proper selection of connection weights and thresholds, each neuron in the output class can have a peak that occurs at a different value. By making the input to the scheduler network the input or output of the whole system, different neurons in the output class will come on depending on the level of the input or output Each of these

scheduler output neurons can be used to turn on or off various neurons in the modeling neural network, and so different models will become active depending on the input or output levels.

It is interesting to note that neural networks which have this kind of peaked response actually occur in nature. In the cerebellum, which among other things is used to perform motor control, an architecture similar to the one shown in Figure 3.3 exists[5]. Neurons in the output class have the characteristic peaked response.

In the next Section, a method for using this kind of a neural network to perform a scheduling task is described. For clarity, the scheduler network is referred to as a single class, made up of neurons which have a peaked response as described in this Section.

3.2.4 Integrated Identification Architecture

Section 3.2.2 described a simple neural network which could implement an arbitrary linear system. Section 3.2.3 described a class of neurons called scheduler neurons which had a peaked response. This Section discusses how these two results can be used together to produce a neural network architecture which is suitable for identification of nonlinear systems.

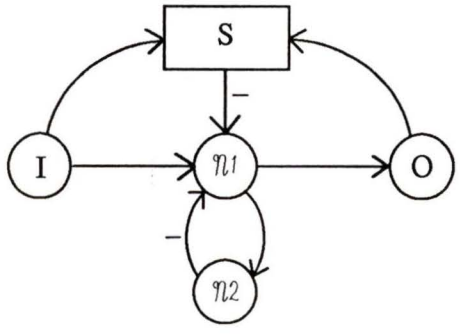

Figure 3.5: An Architecture for System Identification

Figure 3.5 is the architecture which is proposed for identification. Neurons in the scheduler class, which is marked S in the diagram, have a peaked response as described in Section 3.2.3. Figure 3.5 shows that the scheduler class receives input from I and O. The

scheduler is then used to inhibit neurons in the rest of the neural network. The classes marked $\mathcal{N}1$ and $\mathcal{N}2$ are arranged in the format described in Section 3.2.2. Their task is to implement the modeling which that Section described. In order to ensure stability, the feedback connection from $\mathcal{N}2$ to $\mathcal{N}1$ is made inhibitory, and the scheduler class will also only inhibit.

The scheduler class inhibits different neurons in the modeling Section of the neural network different amounts depending on the level of the input or output. Thus, depending on the value of the input and output, different neurons in $\mathcal{N}1$ and $\mathcal{N}2$ will be active. This allows the neural network to take advantage of the type of nonlinearity discussed in Section 3.2.1.

3.3 Summary

This Chapter has discussed an architecture which is suitable for identification of nonlinear systems. A basic analysis of nonlinear systems was presented, showing how nonlinear systems can be thought of as linear systems which had their pole locations vary depending on the level of the inputs and outputs. Then, a neural network was shown which could implement an arbitrary linear system. A neural network was also designed which had a peaked response, and this was explained to be suitable for the task of scheduling various models for different input and output levels. These two neural networks were then combined to produce an architecture which is suitable for identification of nonlinear systems. Thus, a logical explanation for this choice of architecture was presented, based on a basic analysis of nonlinear systems.

Chapter 4 : Experimental Results

4.0 Linear System Identification

In this Section, identification of linear systems is discussed. First-order and second-order linear systems are identified using a stable neural network as described in Chapter 3. Since a linear system of higher order can be described by the superposition of linear systems of first and second order by the method of partial fraction expansion, these two special cases are sufficient to show that a stable neural network can identify linear systems of higher order.

Since the systems to be identified in this Section are linear, it is logical to choose a linear activation function for the neural networks. However, this is not possible since a linear function is not in the neuromime class as described in Section 2.2. Specifically, the condition $f: R^N \rightarrow R_+^M$ is not met since the linear function maps outside of the positive reals.

Neural networks in this Section have an activation function which is described by the relation

$$f(o) = \begin{cases} 0, o < 0 \\ o, o \geq 0 \end{cases} \tag{4.1}$$

This function is of the neuromime class described in Chapter 2, and thus the stability results of Section 2.2 apply to the neural networks presented in this Section. Furthermore, provided that the input to the neural network is biased so that neural activations are always positive, then the activation function is effectively linear. Thus, in the analysis of the neural networks which takes place in this Chapter, the activation function $f()$ is ignored for the sake of clarity and to allow for more insightful analysis of the networks.

4.0.0 First Order Response

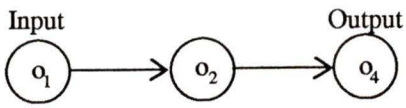

Figure 4.1: Neural Network for First-Order Systems

A general first order system has a response which is described in state-space form as

$$\begin{aligned} \dot{x}_1 &= -a_1 x_1 + b_1 u \\ y &= c_1 x_1 \end{aligned} \tag{4.2}$$

where u is the system input and y is the system output. A simple neural network can emulate this system, as shown in Chapter 3 and repeated here for convenience. The neural network shown in Figure 4.1 has the response

$$\begin{aligned} \dot{o}_2 &= -\tau_2 o_2 + w_{21} o_1 \\ \dot{o}_4 &= -\tau_4 o_4 + w_{42} o_2 \end{aligned} \tag{4.3}$$

The state of o_1 is assumed to be set directly, i.e., it can be considered an input neuron. If the relaxation constant of o_4 is made much smaller than that of the other neurons in the network, then equation (4.3) can be approximated by

$$\begin{aligned} \dot{o}_2 &= -\tau_2 o_2 + w_{21} o_1 \\ o_4 &= \tfrac{w_{42}}{\tau_4} o_2 \end{aligned} \tag{4.4}$$

Equation (4.4) has the same form as equation (4.2), and thus it is seen that with the proper selection of the various weights and time constants, the neural network of Figure 4.1 can emulate any first-order linear system.

In this case, it is trivial to select proper weights and time constants to allow for model following. However, the parameter adjustment equations developed in Chapter 3

can be used to adapt the weights and time constants so that the simple neural network shown here can have the desired model response. It is instructive to use the full training procedure even in this simple example for the sake of illustration. An appropriate training procedure is now outlined.

In this simulation, the first-order linear system to be identified is given by

$$\begin{aligned} \dot{x}_1 &= -5x_1 + 3u \\ y &= 1.5x_1 \end{aligned} \tag{4.5}$$

This system has the step response shown in Figure 4.2. Inspection of equation (4.4) shows that for proper model following, the neural network may have $\tau_1 = 5$, $w_{21} = 3$, and the ratio $\frac{w_{42}}{\tau_4} = 1.5$. However, due to the fact that this is a linear system, the only absolute requirement is $\tau_1 = 5$, and the product $\frac{w_{21}w_{42}}{\tau_4} = 4.5$. This ratio gives some freedom to the actual values of the parameters of the neural network. This is advantageous since it was mentioned earlier that it is desired to have the time constant of neuron o_4 small compared to the other time constants. Thus the variable τ_4 may be initialized to a large value to assist with training.

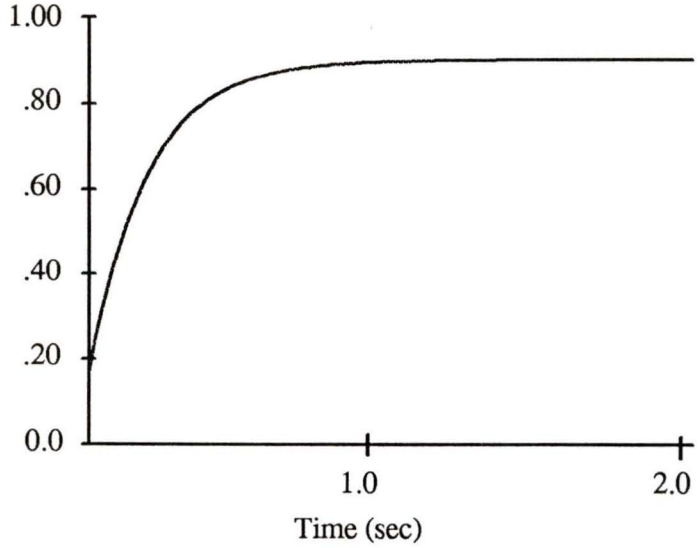

Figure 4.2: Step response for system of equation (4.5)

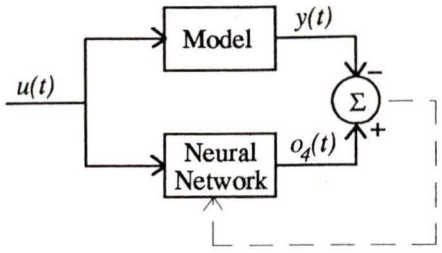

Figure 4.3 : Block diagram for training procedure

A block diagram for the training procedure is shown in Figure 4.3. The neural network has the structure of Figure 4.1 and uses the activation equation of equation 4.1. A random input value $u(t)$ is prepared and applied to the input of the model, and the state of neuron o_1 is also set to the value of this input. Instantaneous values for the states of all the neurons in the neural network, and the value $y(t)$ of the model output, are then calculated using numerical integration. The values $y(t)$ and $o_4(t)$ may be compared, and an error for the cost function is thus generated. The cost function to be used in this case is

$$J(e(t)) = \tfrac{1}{2}e^2(t) \tag{4.6}$$

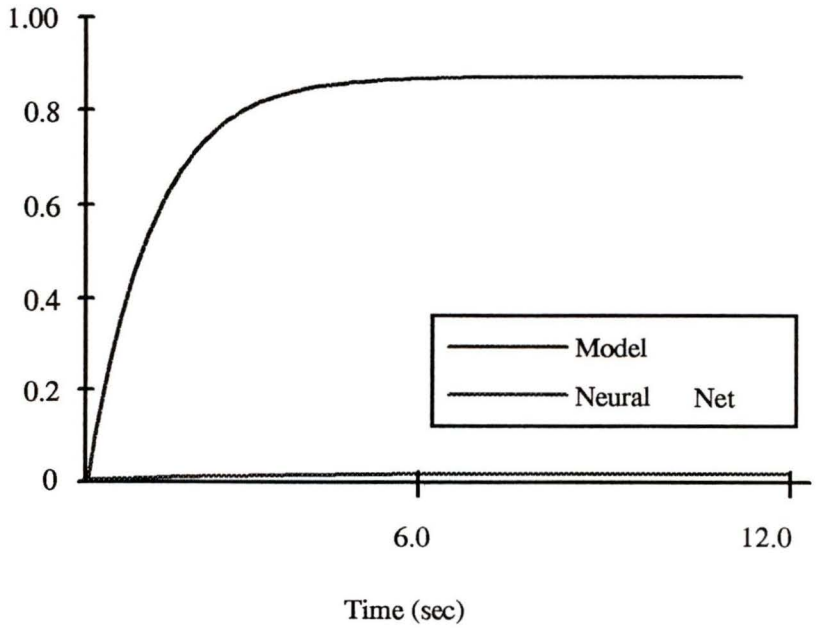

Figure 4.4: Model and Initial Step Response for untrained neural network

The derivatives of the weights and time constants of the neurons o_4 and o_2 with respect to the cost may be calculated using the formulae developed in Section 2.3, and so the various parameters may be adjusted. This procedure is repeated until the error is at an acceptable value.

This training procedure was applied to the neural network shown in Figure 4.1. The weights and time constants were initially set to random values, and this caused the neural network to have the before-training response shown in Figure 4.4. The after training response is shown in Figure 4.5. The weight and time constants are summarized in table 4.1.

	τ_2	τ_4	$\frac{w_{21}w_{42}}{\tau_4}$
Ideal	5.0	∞	4.5
Before Training	0.6975	85.0	.003209
After Training	5.092	80.5	4.57

Table 4.1: Ideal and Trained Parameters for First Order System

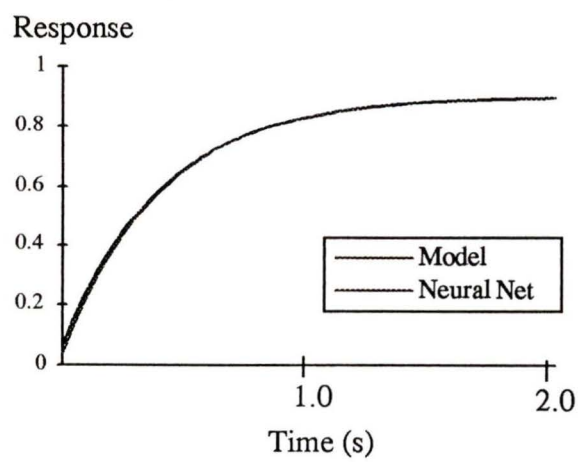

Figure 4.5: After training response for 1st order system.

As can be seen from table 4.1, the parameters converged to values close to the ideal. This is evident in Figure 4.5, which shows that the response of the trained neural network is virtually indistinguishable from the ideal. Figure 4.6 shows that the trained neural network also tracks other inputs, such as noise.

Note that the input has been offset by a large positive value to ensure that the neural network stays in its linear operating range. It is true that in the case of this simple linear example it is possible to say that the semilinear activation function can be replaced by a linear activation function and the input can be both positive and negative. The analysis techniques for linear systems shown in Chapter 1 could be used to prove that the neural network is BIBO stable. However, in the more complicated systems which are shown later in this Chapter, such a conclusion is not possible. Thus, for consistency and to illustrate the technique of identification using stable neural networks, the inputs for the linear systems in this Section will be biased.

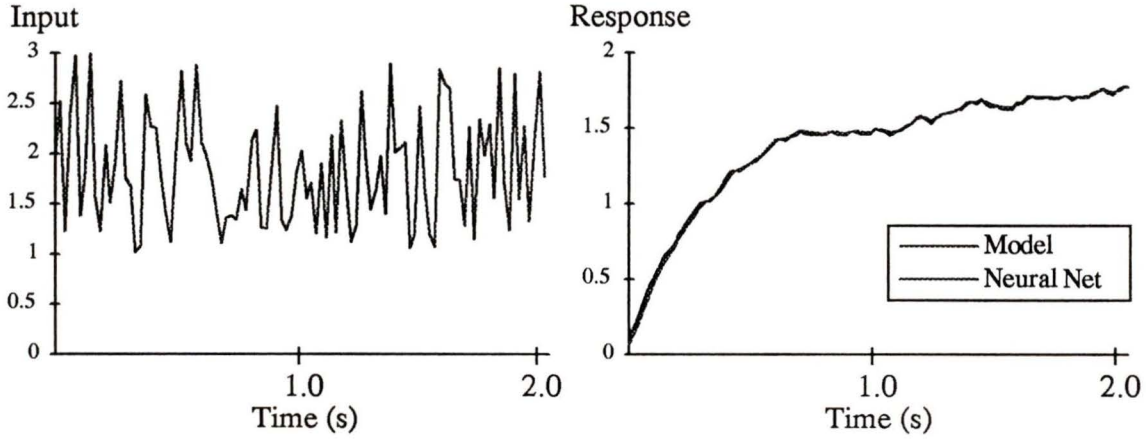

Figure 4.6: Input sequence and Responses for 1st order system

As a measure of the rate of convergence, the mean-square-error between the actual and ideal response was measured throughout the training procedure. An average value for this was calculated every 100 iterations, and this error function is plotted in Figure 4.7

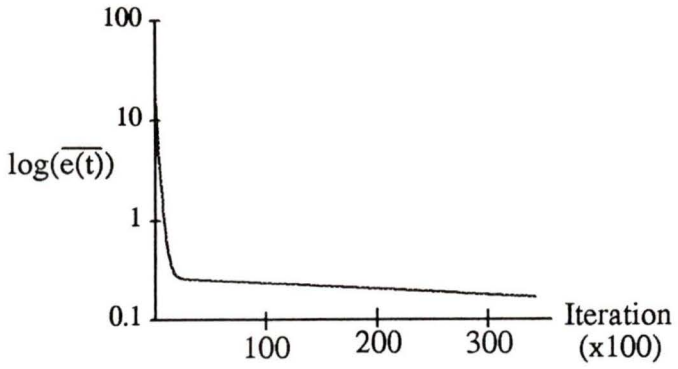

Figure 4.7: Convergence of Error for Identification of 1st order system

As can be seen in Figure 4.7, the error converges very quickly at first, which is due to the fact that the initial response was approximately two orders of magnitude smaller than the model response. After this initial period, convergence is slower. An input sequence of approximately 35000 values was needed to train the neural network. This is a large number of training samples, but it is consistent with other neural network identification techniques[14]. Furthermore, since the neural network is so small, and thus so few parameters are actually being trained, the entire training procedure for this system was finished in a matter of minutes.

4.0.1. Second Order System

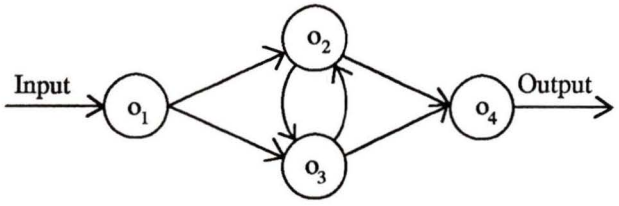

Figure 4.8: Neural Network for Second-Order Systems

A single-input, single-output second order system has a response which is described in state-space form as

$$\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix}=\begin{bmatrix}a_{11} & a_{12}\\ a_{21} & a_{22}\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix}+\begin{bmatrix}b_1\\ b_2\end{bmatrix}u$$
$$y=\begin{bmatrix}c_1 & c_2\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix} \tag{4.7}$$

where u is the system input and y is the system output. The neural network shown in Figure 4.8 can implement this system, as was shown in Chapter 3. The neural network shown in Figure 4.8 has the response

$$\begin{bmatrix}\dot{o}_2\\ \dot{o}_3\end{bmatrix}=\begin{bmatrix}-\tau_2 & -w_{23}\\ w_{32} & -\tau_3\end{bmatrix}\begin{bmatrix}o_2\\ o_3\end{bmatrix}+\begin{bmatrix}w_{21}\\ w_{31}\end{bmatrix}o_1$$
$$o_4=\tau_4^{-1}\begin{bmatrix}w_{42} & w_{43}\end{bmatrix}\begin{bmatrix}o_2\\ o_3\end{bmatrix} \tag{4.8}$$

provided that the time constant of o_4 is made much smaller than that of the other neurons in the network. The state of o_1 is assumed to be set directly, i.e., it can be considered an input neuron.

Equation (4.7) has the same form as equation (4.8), although it is seen that there are some restrictions on the polarity of the values in equation (4.8) that do not exist in equation (4.7). This is not a problem, however, since as was shown in Chapter 1, a similarity transform can be used to convert from one form to another. As was shown in Chapter 3, the system of equations (4.8) can have any desired stable eigenvalues, and thus in order to make the neural network have the same responses as the model it would merely be necessary to find a similarity transform that allows for the desired polarities.

Although this is a possible design procedure, the parameter adjustment equations developed in Chapter 3 can be used to adapt the weights and time constants so that the neural network shown here can have the desired model response. A similar training procedure to the one used in Section 4.0.0 will be used here.

In this simulation, the linear system to be identified is given by

$$\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix}=\begin{bmatrix}0 & 1\\ -10 & -5\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix}+\begin{bmatrix}0\\ 3\end{bmatrix}u$$
$$y=\begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix} \tag{4.9}$$

which is equivalent to the transfer function $H(s)=\dfrac{3}{s^2+5s+10}$. The eigenvalues for the system matrix of equation 4.9 are complex conjugate values located at -2.5 +j1.937 and -2.5-j1.937.

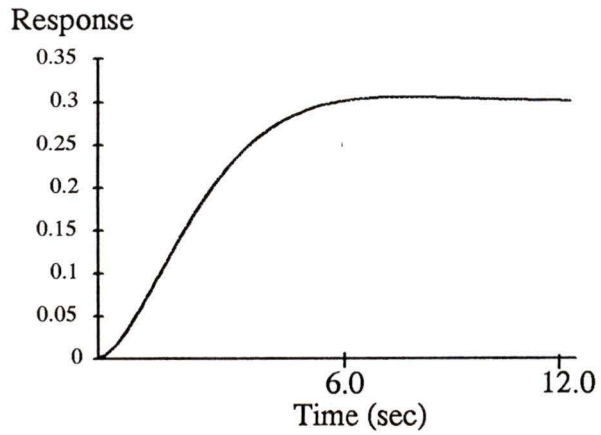

Figure 4.9: Step response for system of equation (4.9)

This system has the step response shown in Figure 4.9. As was the case for training the first-order neural network, a random sequence $u(t)$ is used for training. As discussed in Section 2.0.4, a random sequence is the most persistently exciting signal available. Once again, the cost function to be used is

$$J(e(t))=\tfrac{1}{2}e^2(t) \tag{4.10}$$

The value τ_4 was initialized to a large value to ensure that the output neuron followed its inputs, and its own dynamics did not affect the overall response. The initial step response for the neural network is shown in Figure 4.10.

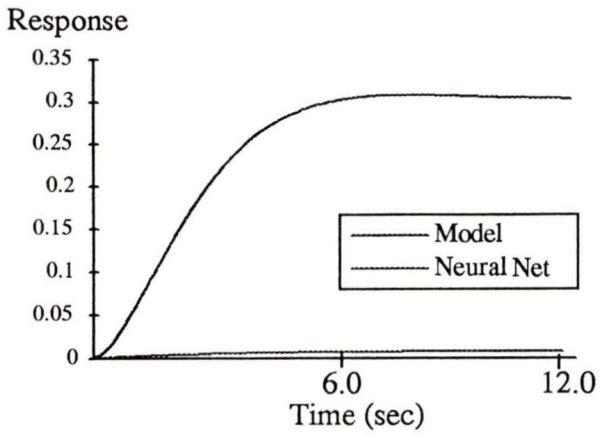

Figure 4.10: Initial Step Response for untrained neural network

This training procedure was applied to the neural network shown in Figure 4.8. After training, the neural network had the step response shown in Figure 4.11. The step response for the model of equation 4.9 is also shown in Figure 4.11, showing that the two responses are virtually identical.

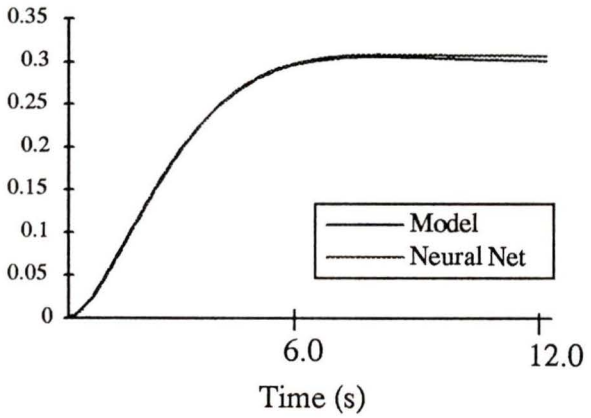

Figure 4.11: After training response for 2nd order system.

The effective state-space matrices of the trained neural network are compared to the model in table 4.2.

Quantity	Model	Trained Neural Network
System Matrix A	$\mathbf{A}_m = \begin{bmatrix} 0 & 1 \\ -10 & -5 \end{bmatrix}$	$\mathbf{A}_n = \begin{bmatrix} -.492 & 1.08 \\ -7.75 & -4.52 \end{bmatrix}$
Eigenvalues	$-2.5 \pm j1.937$	$-2.53 \pm j1.94$
Input Matrix B	$\mathbf{B}_m = \begin{bmatrix} 0 \\ 3 \end{bmatrix}$	$\mathbf{B}_n = \begin{bmatrix} .234 \\ 2.95 \end{bmatrix}$
Output Matrix C	$\mathbf{C}_m = [1 \quad 0]$	$\mathbf{C}_n = [.793 \quad -.0512]$

Table 4.2: Comparison of values for Neural Net and Model

The values in table 4.2, with the exception of the line which shows the eigenvalues of the system matrix, do not reveal very much. However, noting that the eigenvalues are roughly equal, an approximate similarity transform can be obtain to compare the two systems. Note that

$$\mathbf{A}_n = \mathbf{S}_1 \mathbf{D}_1 \mathbf{S}_1^{-1},$$
$$\mathbf{S}_1 = \begin{bmatrix} .235 - j.245 & .235 + j.245 \\ j.9407 & -j.9407 \end{bmatrix}, \text{ and} \tag{4.11}$$
$$\mathbf{D}_1 = \begin{bmatrix} -2.53 + j1.94 & 0 \\ 0 & -2.53 - j1.94 \end{bmatrix}$$

and

$$\mathbf{A}_m = \mathbf{S}_2 \mathbf{D}_2 \mathbf{S}_2^{-1},$$
$$\mathbf{S}_2 = \begin{bmatrix} .185 - j.238 & .185 + j.238 \\ j.954 & -j.954 \end{bmatrix}, \text{ and} \tag{4.12}$$
$$\mathbf{D}_2 = \begin{bmatrix} -2.5 + j1.937 & 0 \\ 0 & -2.5 - j1.937 \end{bmatrix}$$

It is possible to use the approximation $\mathbf{D}_1 \cong \mathbf{D}_2$ to obtain the similarity transform

$$\begin{aligned} \mathbf{A}_m &\cong \mathbf{S}_2\mathbf{S}_1^{-1}\mathbf{A}_n\mathbf{S}_1\mathbf{S}_2^{-1} \\ &= \mathbf{S}^{-1}\mathbf{A}_n\mathbf{S}, \\ \mathbf{S} &\cong \begin{bmatrix} 1.274 & .0618 \\ 0 & .9866 \end{bmatrix} \end{aligned} \tag{4.13}$$

Using this similarity transform, a more meaningful comparison of the two systems is possible. Table 4.3 summarizes this comparison. Table 4.3 reveals that the model and trained neural network are in fact very similar representations, which is expected from Figure 4.11.

Quantity	Model	Trained Neural Network
System Matrix A	$\mathbf{A}_m = \begin{bmatrix} 0 & 1 \\ -10 & -5 \end{bmatrix}$	$\mathbf{S}^{-1}\mathbf{A}_n\mathbf{S} = \begin{bmatrix} -.065 & 1.08 \\ -10.07 & -5.01 \end{bmatrix}$
Input Matrix B	$\mathbf{B}_m = \begin{bmatrix} 0 \\ 3 \end{bmatrix}$	$\mathbf{S}^{-1}\mathbf{B}_n = \begin{bmatrix} .043 \\ 2.91 \end{bmatrix}$
Output Matrix C	$\mathbf{C}_m = [1 \quad 0]$	$\mathbf{C}_n\mathbf{S} = [.999 \quad -.005]$

Table 4.3: Comparison of values for Neural Net and Model using Similarity Transform

The error function was obtained by averaging the instantaneous square difference between the neural net and the model over 100 data points. The resulting graph is shown in Figure 4.12. Note that the second order system converges in approximately an order of magnitude longer than the first order system did. One possible explanation for this is that the error surface in the parameter space is more complicated for the second order system than it is for the first order system. This is due to the fact that the semi-linear activation function can cause the neural network to operate outside of its linear range. In the first-order system, a positive input implied that the state of o_2 would always be positive, and

hence the semi-linear function behaved exactly like a linear function. However, with a second-order system, it is possible to have the case were one of the state variables is negative in response to a positive input, even a constant positive input. If this is the case in the neural network, for instance if o_2 becomes negative then the output neuron will leave its linear range, and the semi-linear function will cause the output neuron to shut off the input from o_2. Also, neuron o_3, which receives also input from o_2, will no longer receive that input. This effectively results in a first-order linear system until o_2 returns to a positive value.

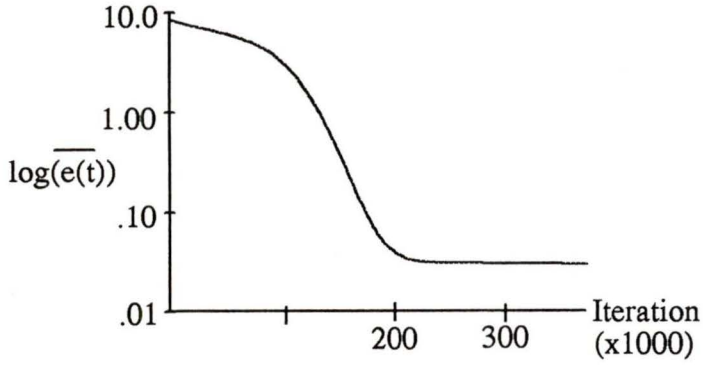

Figure 4.12: Convergence of Error for Identification of 2nd order system

The result of this is that during the training procedure, the neural network must first find a region in parameter space where both neurons o_2 and o_3 can operate in their positive region. It also must keep the parameters within this region for the duration of the training procedure. This is one possible explanation for the slower convergence of this system than the first order example.

It should be mentioned that during the running of the simulation, the neural network would occasionally be trained into a local minimum of the cost function where one of the neurons o_2 or o_3 did indeed stop operating outside of the linear region. When this happened, the gradient descent parameter adjustment rule could not bring the neural network back into the linear range. The result was a trained response which was far from the desired response. When this happened, the simulation was restarted with a different set

of initial parameters and a different random input sequence. This problem could also be handled by adding a term to the cost function which places a heavy value on keeping o_2 and o_3 positive. For instance,

$$J(e) = \tfrac{1}{2}e^2 + k\exp(-a(o_2 + o_3)) \tag{4.14}$$

As o_2 and o_3 approach 0, the second term becomes important and acts to keep the neurons in their operating range. This was not done in the simulations presented in this thesis, since an acceptable solution was found to be merely restarting the program.

This Section has presented an example of how a stable neural network can implement a second order linear system. Since linear systems of higher order can be represented by a parallel combination of first and second order systems through the use of partial fraction expansion, this Section and the previous Section are sufficient to show that a stable neural network can implement linear systems of higher order. The rest of this Chapter will deal with more complicated systems.

4.1 Identification of Nonlinear Systems

This Section deals with training a neural network to have a nonlinear response. In Section 4.1.0., a neural network is trained to have a peaked response. This is desired for scheduler neurons. In Section 4.1.1, a neural network is trained to follow a simple nonlinear system.

4.1.0 Training of the Scheduler Class

This Section describes the procedure for training a neuron to have a peaked response, as is desired for a neuron to perform the job of scheduling as discussed in Chapter 2.

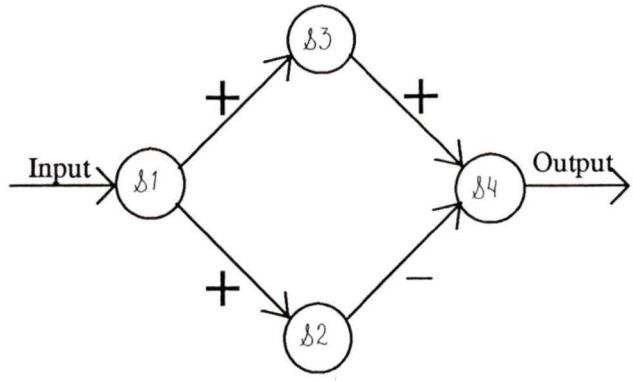

Figure 4.13: Architecture for Scheduler class

Figure 4.13 shows the architecture which was developed in Chapter 3 for a peaked-response neural net, where the theory of operation was explained. This neural network was used here, with one neuron in each class. The activation function which was used for these neurons was

$$f(o)=\begin{cases}0, & o<0\\ \dfrac{1}{1+e^{-\sigma o}}-.5, & o\geq 0\end{cases} \tag{4.15}$$

This is illustrated in Figure 4.14.

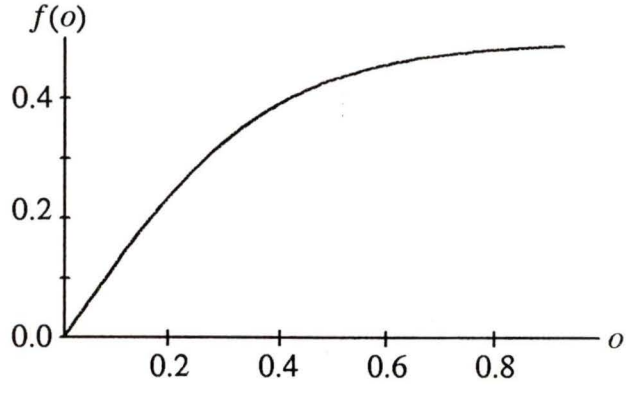

Figure 4.14: Activation function used for scheduler network

This function is similar to the one shown in Figure 3.4 as it has lower and upper thresholds. The weights in the neural network were originally set to random values, with

the polarity as indicated in Figure 4.13. Weight clamping was used to ensure that the correct polarity was maintained throughout the training procedure.

The input-output response can be obtained by applying an input step of height a, measuring the steady-state value b for neuron o_4, and plotting b vs. a. The initial response of the neural network, obtained using this procedure, is shown in Figure 4.15.

The training procedure consisted of applying a step of a random height between 0.0 and 0.5, and using numerical integration to obtain the steady state response of the output class. The desired, or model, response is the idealized peaked response shown in Figure 4.16. Clearly the simple neural network of Figure 4.13 cannot be expected to have such an idealized response without using a discontinuous activation function, nor is it clear that such a response would even be desirable for use as an actual scheduler. This is because a smooth transition from one model to the next is desired, and such a rapid cutoff would likely cause irregularities during the crossover. However, the qualitative properties of the model should serve to train the neural net.

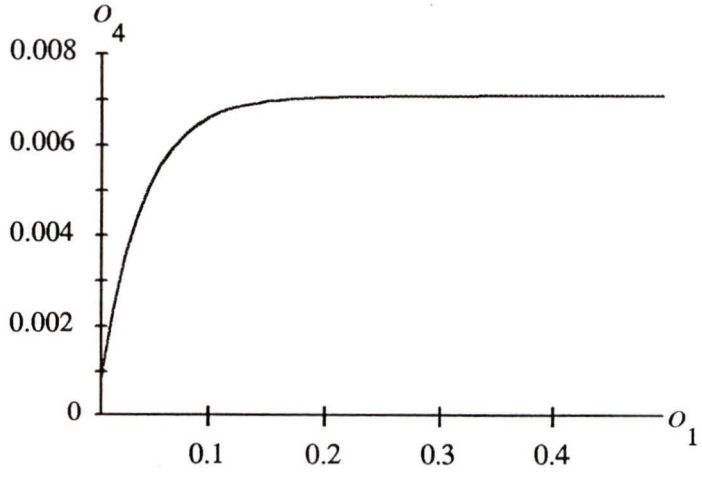

Figure 4.15: Input-output response for untrained scheduler network

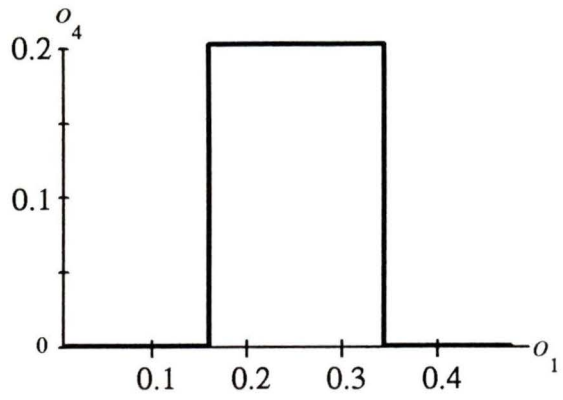

Figure 4.16: Ideal input-output response for model scheduler

Once again, the cost function used is the quadratic cost $J(e(t)) = \frac{1}{2}e^2(t)$. For this neural network, the only parameters which were adjusted were the network weights. This is because the qualitative discussion given in Section 3.2.3 argues that the desired peaked response can be achieved just by adjusting the weights to certain relative levels. After 100 000 weight adjustments, the neural network had the response shown in Figure 4.17. The values for the trained weights are shown in Figure 4.18.

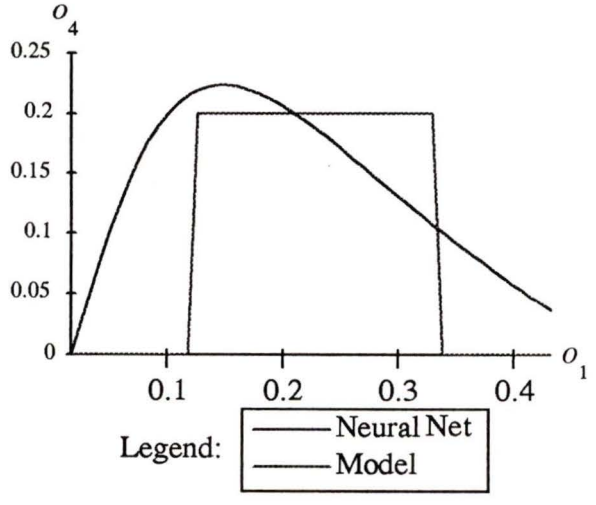

Figure 4.17: Trained Neural Net and Idealized peaked response

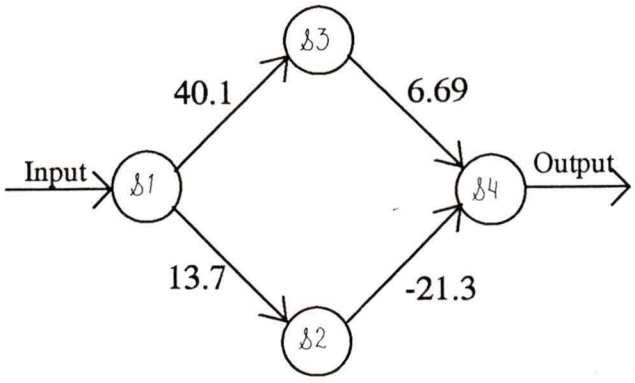

Figure 4.18: Values for trained weights

Figure 4.17 shows that the neural network has developed a response which is indeed peaked. Although not as steeply peaked as the model, this type of response functions sufficiently well for the job of scheduling. Figure 4.18 reveals that the training procedure has produced weights which are in accordance with the qualitative theory of operation given in Chapter 3.

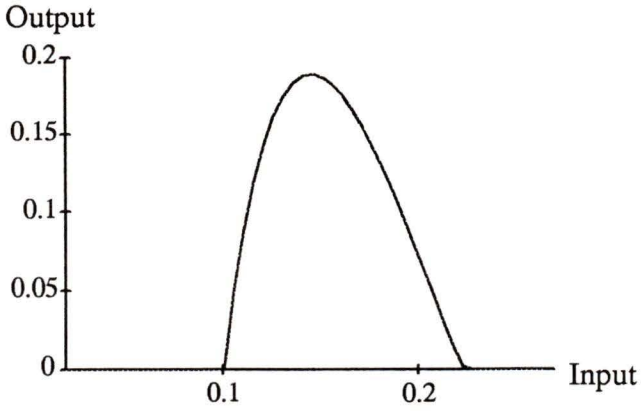

Figure 4.19: Input-Output response demonstrating alternate placement of peak

In order to be useful for scheduling, it is necessary that the peak of the output neuron can be placed in any desired location. This is in fact quite easy. The response shown in Figure 4.19 was obtained using the identical network and connection weights as used in the previous example, with the activation function

$$f(o)=\begin{cases}0, & o<.1\\ \dfrac{1}{1+e^{-\sigma(o-.1)}}-.5, & o\geq .1\end{cases} \tag{4.16}$$

This activation is the a shifted version of the function used in the previous example. Note that the amount of the shift, which is 0.1, corresponds to the point at which the output neuron first turns on. Thus the position of the peak can be controlled in a straightforward manner.

4.1.1 A Simple Nonlinear System

In this Section, a simple nonlinear system is identified using a neural network. The system to be identified is governed by the relation

$$\dot{y}=u-\frac{y}{1+4y^2} \tag{4.17}$$

This example was discussed in Chapter 3. When the input causes y to vary significantly, then it is possible to think of this system as a first order linear system whose pole varies with the level of the output. This is not an exact description of the behaviour of the system. In fact, this system exhibits non-global stability. The relevance here is that the input must be kept within a certain range in order to avoid unstable behaviour. A brief discussion of this will now be given.

Consider a step input of the form

$$u(t)=\begin{cases}0, t<0\\ k, t\geq 0\end{cases} \tag{4.18}$$

The system described by 4.17 will obtain a maximum value for $y(t)$ which can be calculated by setting $y'=0$. This shows that the maximum value is

$$y_{\max}=\frac{1\pm\sqrt{1-16k^2}}{8k} \tag{4.19}$$

For real inputs, the state must remain real, and thus a maximum value can only occur if $k < \frac{1}{4}$. This limiting value of k gives a steady state value of $y = \frac{1}{2}$. When k becomes larger and y surpasses the value of $\frac{1}{2}$, equation (4.17) shows that the differential equation becomes unstable. This is because $\frac{y}{1+4y^2} < \frac{1}{4} \; \forall \, y > \frac{1}{2}$, and so (4.17) shows that the derivative $\dot{y}$ will always be positive. Thus the state will continue to grow with time.

The conclusion from this is that the input must be kept below $\frac{1}{4}$. Provided this condition is met, the system will behave like a first order system that has a level-dependent or roving pole. To take advantage of this type of nonlinearity, an architecture similar to the one of Figure 4.20 was proposed in Chapter 3. In this Figure, the labels I and O refer to the input and output of the system, and $\mathcal{N}1$ refers to a class of neurons. The block marked S represents the scheduler class, which was discussed in Section 4.1.0. Strictly speaking, this is not a single class but it is discussed as a single class for the purpose of clarity.

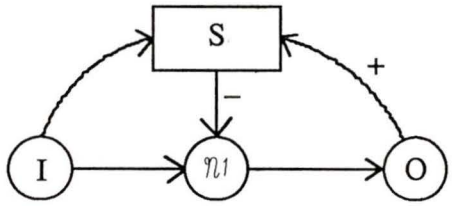

Figure 4.20: An Architecture for System Identification

Note that the neural network shown in Figure 4.20 does not include a second class of neurons $\mathcal{N}2$ connected in a feedback loop with $\mathcal{N}1$ as was shown in Chapter 3. This decision was made since it is known that the model to be identified here is a first order roving pole system. As was shown in Section 4.0, which discussed linear systems, the feedback loop is required for modeling second order effects. Since this system does not have any such effects, the loop was deleted.

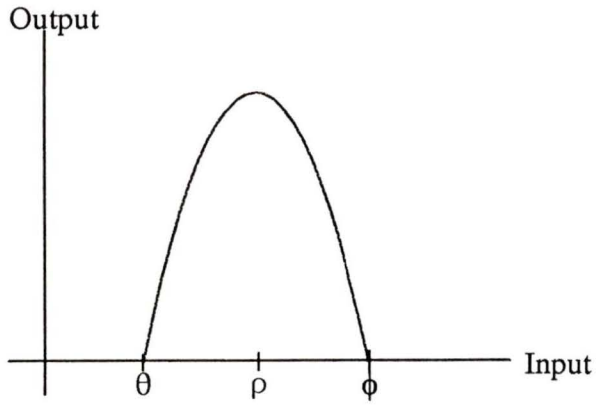

Figure 4.21: Input-Output response of pre-trained scheduler neurons

The training procedure will now be described. $\mathcal{N}1$ contained five neurons, and there were five neurons in the scheduler class. Rather than train the scheduler class, a pre-trained scheduler was used. This consisted of 5 special-purpose neurons which had an output as shown in Figure 4.21. Each neuron had an 'on' region which overlapped with the 'on' region of its neighbours. For instance, the value ϕ of neuron 1 was equal to the value ρ for neuron 2 and θ for neuron 3. This allows for a smooth transition between operating models.

The input consisted of a combination of a step of random height added to white noise. An instantaneous value for the input was calculated and presented to the neural network. The instantaneous error was then determined by calculating the output of the neural net and of the model. The cost function used was once again $J(e(t)) = \frac{1}{2}e^2(t)$.

This training procedure was continued for over a million iterations. After training was complete, this simple neural network was able to follow the response to the input shown in Figure 4.22. The response of the neural network and the actual model are shown in Figure 4.23.

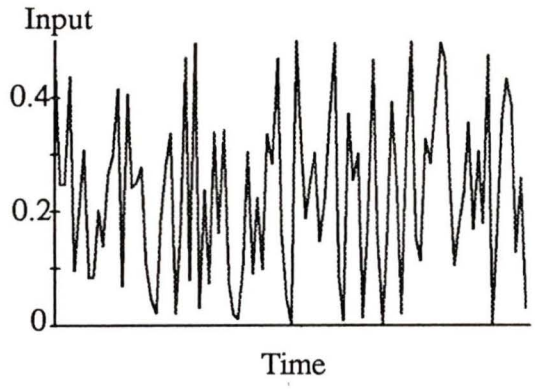

Figure 4.22: Input to Model and Neural Network:

As Figure 4.23 shows, good model following was eventually obtained. However, the training procedure was again approximately an order of magnitude larger than training for a second order linear system. This took approximately two days to run on a Sparcstation.

Although this training time is excessive, it should be noted that this is *offline* training time. This means that if this neural network were to be used to identify a real system, the training procedure is done offline and does not actually require any downtime. Furthermore, due to the small size of this neural network, the online or running speed, i.e., the speed at which the neural network calculates the output at each time interval, is very fast. Thus although the neural network takes a very long time to train, once it has been trained it provides a fast model.

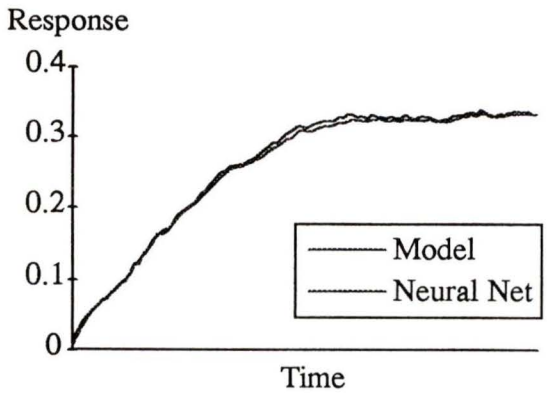

Figure 4.23 : Model and Neural Net Responses

This Section has shown that a stable neural network using a training procedure as outlined in Chapter 3 can identify simulated nonlinear systems. This concludes the simulated systems which the neural network will identify. The next Section discusses the use of a neural network in identifying a real system.

4.2 Identification of a PUMA-560 Robot Arm

In this Section, the neural network is used to model a two-link PUMA-560 Robot. This is effectively a two-input, two-output system, with the two inputs being the two control voltages for the joint motors, and the outputs being the two joint angles. In this Section, only one output is identified. The reason for this is that the identification procedure for the second angle is identical to that for the first. Reducing the size of the output vector can reduce the size of the neural network required for modeling, thereby reducing training time.

4.2.0 Selection of Training Input

For the identification process, a control input needs to be prepared for each motor. The Section on persistent excitation showed that white noise was the preferred input, but this kind of an input is fatiguing for the motors. In order to determine an effective input, some analysis of the robot dynamics is required.

A two-link robot arm is known to have its dynamic response governed by the differential equation

$$\mathbf{H(q)\ddot{q} + h(q,\dot{q})\dot{q} + F\dot{q} + g(q) = \Phi v} \qquad (4.20)$$

where the state $\mathbf{q}$ contains the angle θ_1 that the first link makes with the vertical and the angle θ_2 that is formed between the two links; $\mathbf{H(q)}$ is the 2x2 inertial matrix; $\mathbf{h(q,\dot{q})}$ models the Coriolis and centripetal forces; $\mathbf{F}$ is the friction matrix; $\mathbf{g(q)}$ represents the gravitational torque; and Φ is the voltage-to-torque conversion matrix [3].

Equation (4.20) is deceptive since it makes the dynamics appear linear. However, note that the matrices $\mathbf{H}(\mathbf{q})$ and $\mathbf{h}(\mathbf{q},\dot{\mathbf{q}})$ actually depend on the state, as does the torque vector. This of course makes the system nonlinear.

If this were a linear system, equation (4.20) shows that there would be 14 unknown constants in all the matrices. This would require an input signal which was persistently exciting of order 14 at least for proper identification. Since the model is nonlinear, this analysis is not completely accurate, but it does give an idea of what kind of input is required.

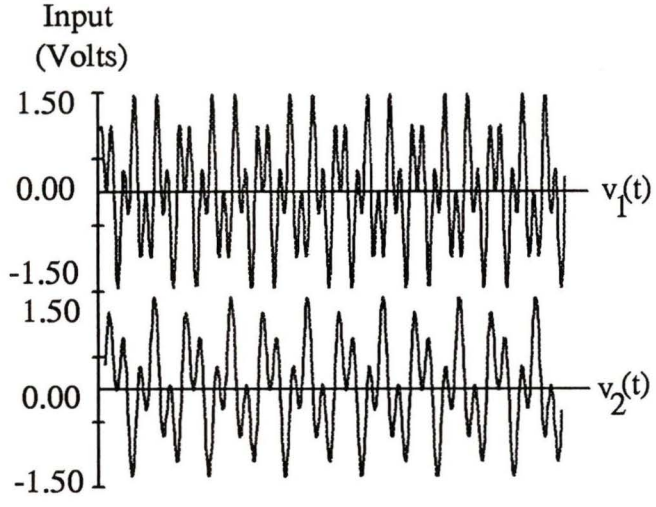

Figure 4.24: Control Voltages

Each input sequence was designed to be a combination of 4 sinusoids. Since a sinusoid is persistently exciting of order 2, each input was persistently exciting of order 8, and thus the input vector was PE of order 16. The control inputs are shown in Figure 4.24.

4.2.1 Classical Identification

All the matrices in equation (4.20) rely on many machine-specific factors, such as dimensions, weights, inertia, and joint friction. Due to the complicated manner in which these factors are combined to produce the matrices, classical identification techniques make it difficult to obtain an accurate model.

One way to mitigate these difficulties is to obtain direct measurements of as many variables as possible. This was done for a PUMA-560 robot. Lengths, masses, and inertias were obtained through direct measurement [6]. The variables which could not be easily measured directly were the matrices **F** and Φ. This represented four unknown scalars in total.

Classical RLS parameter estimation was used to identify these four variables, and the final response to the input vector shown in Figure 4.24 is shown in Figure 4.25. Figure 4.25 also shows the measured response to the input.

Although Figure 4.25 shows that there was some prediction error, it was found [3] that this model was accurate enough to allow for an extremely accurate controller design when this model was used for closed loop control. The fact that the model deviates from the actual response underlies the difficulty in identifying complex systems using traditional model based methods.

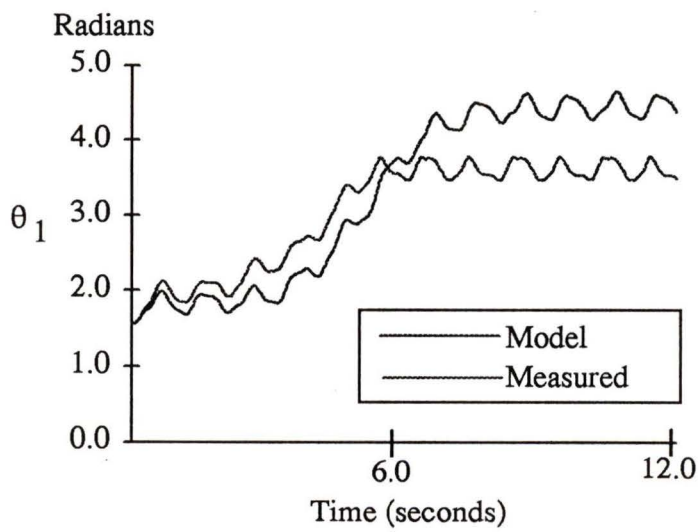

Figure 4.25: Classical Model and Measured Response to Input

4.2.2 Identification Using a Stable Neural Network

The identical neural network that was used to identify the simple simulation of equation 4.17 was trained to identify the dynamic response for θ_1, the angle that the first

link makes with the vertical. Both $v_1(t)$ and $v_2(t)$ were used as inputs to the system. It was decided to use the identical neural network as was used in Section 4.1.1 in order to test the flexibility of the neural network, since the two systems are certainly extremely different.

As in Section 4.1.1, Class $\mathcal{N}$ and the scheduler class each contained 5 neurons. The scheduler class was 'pretrained' for this procedure as it was in Section 4.1.1. The training procedure consisted of applying an input vector to the neural network and calculating the output. This was compared to the measured output to produce an error for the cost function, which was once again the simple quadratic cost.

Convergence of the response was rather slow. Several million training iterations were required to obtain the response in Figure 4.26. However, due to the small size of the neural network, good results were obtained after two days of training on a Sparcstation, and continued training for two more days afterwards continued to improve the response to produce the results seen in Figure 4.26. Once again it should be noted that this is the offline training time. The running speed of the neural network is still very fast. Furthermore, no design time was needed to develop a complex model such as equation 4.20, and the laborious process of physical parameter measurement is avoided.

Comparison of Figures 4.25 and 4.26 reveal that the neural network was able to track the measured response more closely. The most likely reason for this is some sort of error in the direct measurement of the PUMA parameters. In a nonlinear system such as this one a small measurement error can produce a large modeling error. However, since the identified model is rarely used in an open-loop situation, but rather in a closed loop control application, the error is tolerable.

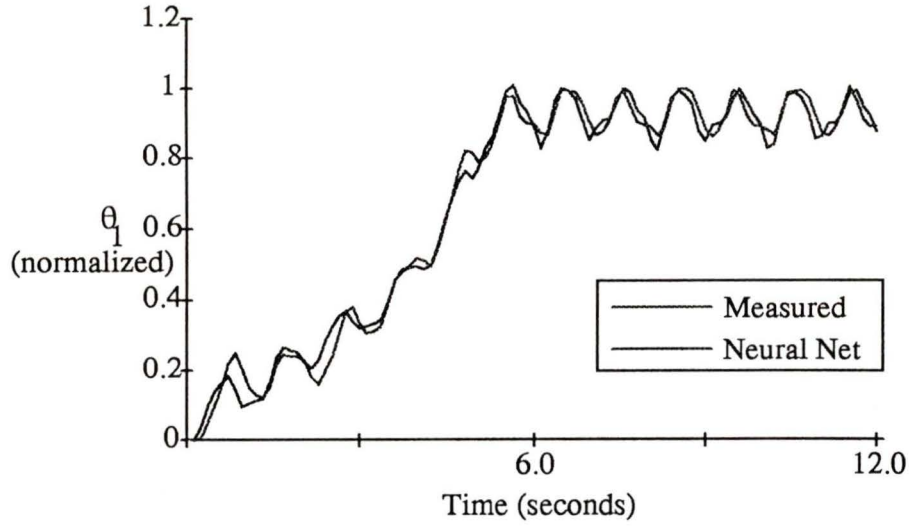

Figure 4.26: Measured and Neural Net Response to Input shown in Figure 4.24

4.3 Identification of a Boat

To further test the neural network's flexibility, the identical neural network which was used to model the robot arm of the previous Section was trained to model a motorized boat moving through the ocean with constant velocity. The data for this experiment was collected by Alex Andekian for use in [21].

For the purposes of this experiment, the boat was treated as a SISO system, with the rudder angle being the input and the heading being the output. Extensive work has been done to produce accurate models for marine craft. In [21], a first-order linear system is

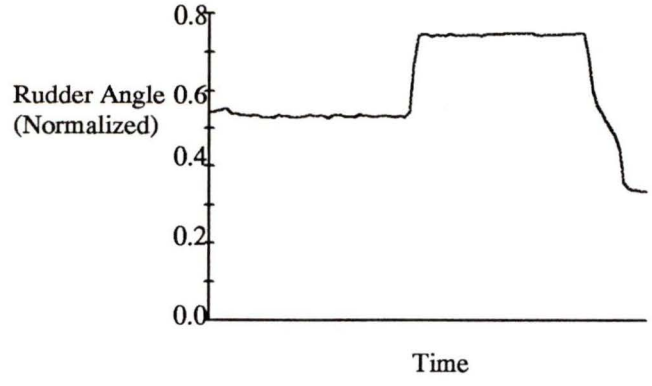

Figure 4.27: Rudder Angle for Training Run

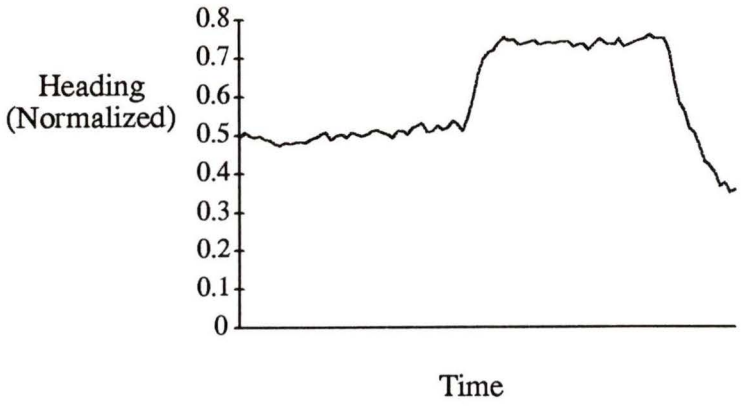

Figure 4.28: Boat heading for Training Run

used to approximate the boat.

Figure 4.27 shows the input training sequence which was used. This is equivalent to approximately 2 minutes of data collected from the boat. Figure 4.28 shows the response which the onboard gyroscope yielded.

The identical neural network used to identify the robot was used here. The training method consisted of applying the input of Figure 4.27 to the neural network, calculating its response, and using the measured heading shown in Figure 4.28 to generate an error signal for weight adjustment. Figure 4.29 shows the response of the trained neural network to the training data. As can be seen, the neural network follows the measured response quite closely.

It should be noted that the neural network does not exactly follow the measured data. Rather, the response of the neural net appears to be a 'smoothed', or low-pass filtered, version of the actual response. In reality, this is probably because of the fact that inherent in this and any other system which links to the physical world is some measurement noise. Since the neural network is a deterministic model, the noise process, which is unpredictable by such a model, will not appear in the output.

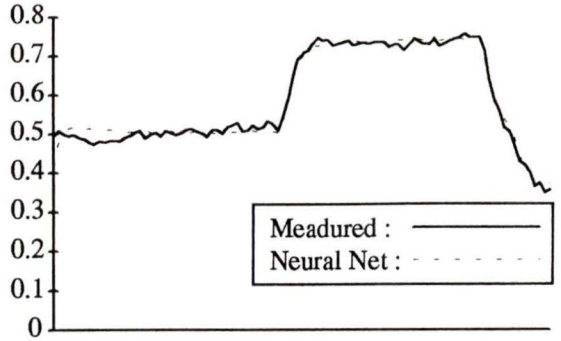

Figure 4.29: Measured and Trained Neural Net Response to Input shown in Figure 4.27

To test if the neural network had completely identified the boat at this particular speed through the water, a longer run of data was used. This consisted of approximately 12 minutes of collected data. The weights which had been developed on the shorter training run were used. Figure 4.30 shows that the neural network had indeed developed a good model of the boat since good tracking was obtained throughout the longer test run.

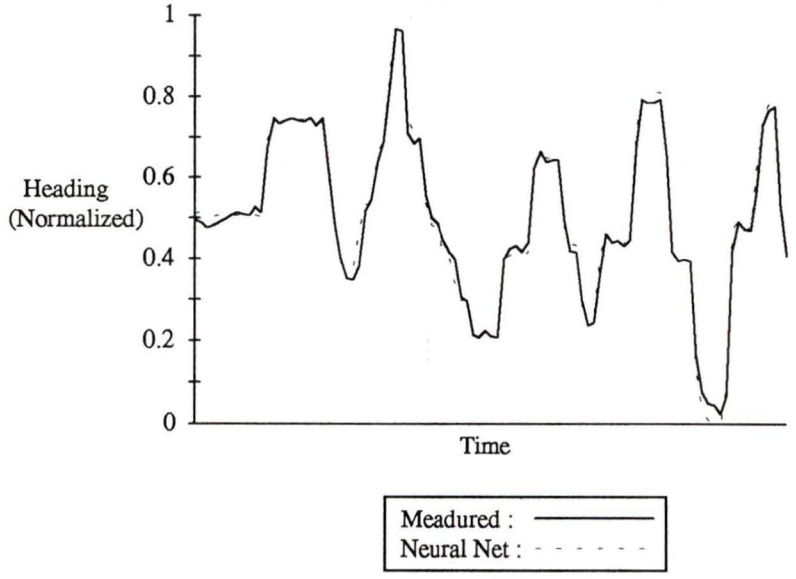

Figure 4.30: Measured and Neural Net Response to 12 Minute Test Run

Bibliography

[1] B. Armstrong, O.Khatib, and J. Burdick, "The Explicit Dynamic and Inertial Parameters of the PUMA-560 Arm", *IEEE International Conference on Robotics and Automation*, 1986, pp. 510-518

[2] Karl Johan Astrom and Bjorn Wittenmark, Adaptive Control, Addison-Wesley, 1989.

[3] S.A. Billings, "Identification of Nonlinear Systems", in Nonlinear System Design, Peter Peregrinus Ltd., 1984.

[4] John J. Craig, Introduction to Robotics, Addison-Wesley, 1988.

[5] N. Dimopoulos, "A Study of the Asymptotic Behaviour of Neural Networks," *IEEE Transactions on Circuits and Systems*, Vol. 36, No.5, pp. 687-694, May 1989.

[6] M. Erlic, M.A.Sc. Thesis, University of Victoria, Victoria B.C., 1990.

[7] R. Fletcher, Practical Methods of Optimization, John Wiley & Sons, Ltd., 1987.

[8] Bernard Friedland, Control System Design: An Introduction to State-Space Methods, McGraw-Hill Inc., 1986.

[9] Morris W. Hirsch and Stephen Smale, Differential Equations, Dynamical Systems, and Linear Algebra, Academic Press Ltd., 1974.

[10] Kurt Hornik, Maxwell Stinchcombe, and Halbert White, "Mulitlayer Feedforward Networks are Universal Approximators", Dept. Economics, University of California, San Diego, discussion paper, June, 1988.

[11] Teuvo Kohonen, Self-Organization and Associative Memory, Springer-Verlag, 1984.

[12] Roland E. Larson and Bruce H. Edwards, Elementary Linear Algebra, D.C. Heath and Company, 1988.

[13] Richard P. Lippmann, "An Introduction to Computing with Neural Nets", *IEEE ASSP Magazine*, April, 1987, pp. 4-22.

[14] K.S. Narendra and K. Parthasarathy, "Identification and Control of Dynamical Systems Using Neural Networks," *IEEE Transactions on Neural Networks*, Vol. 1, No. 1, pp.4-27, March 1990.

[15] Charles L. Phillips and H. Troy Nagle, Jr., Digital Control System Analysis and Design, Prentice-Hall Inc., 1984

[16] Peter V. O'Neil, Advanced Engineering Mathematics, Wadsworth, Inc., 1983

[17] Rumelhart and J.L. McClelland, Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1 and 2, MIT Press, 1986

[18] Louis L. Scharf, Statistical Signal Processing : Detection, Estimation, and Time Series Analysis, Addison-Wesley Publishing Company Inc., 1991.

[19] B. Widrow and M. Lehr, "30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation," *IEEE Proceedings, Special Issue on Neural Networks*, Vol. 78, No. 9, Sept. 1990.

[20] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi, "Optimization by Simulated Anealing," *Science*, Vol. 220, No. 4598, May 13, 1983.

[21] A. Andekian, M.A.Sc. Thesis, University of Victoria, Victoria B.C., 1993.

[22] G. Cybenko, "Approximation by superpositions of a sigmoidal function," Mathematics of Control, Signals and Systems, Vol 2., No. 4, pp. 303-314, 1989.

Appenidx A - Program Listing

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "chris.h"
#include "makenet.h"

#define FILECRASH 1
#define FILESUCCESS 0
#define neur(a,b) ( (struct neuron *)( (int)(Layer[(a)].data) +
(b)*Layer[(a)].neuronsize))
#define EOL ((struct neuron *) NULL)
#define SLOPECONST 1  /* used in sigmoid function to determine
slope */
#define NEURPOLE 5.0
#define DRYRUN FALSE     /* used for turning on/off certain
functions for
                        a real training run                    */
#define SCHEDULER_LAYER 1
#define MAXINPUT .6

#define WEIGHT_POS 1
#define WEIGHT_NEG -1
#define WEIGHT_ANY 0

#define SINEINPUT 1
#define STEPINPUT 2
#define NOISEINPUT 3

#define NODERIVRESET 0
#define DERIVRESET 1
/*-------------------------------------------------------------------------*/
struct axons {                  /* An axon points to a presynaptic
*/
        struct neuron  *presynaptic;/* neuron, and has a coupling
weight */
        double weight;
        double tempweight;
        double maxinit;
        double tempdelta;   /* for delta in backprop              */
        double temppsi;         /* for psi in backprop
    */
        int polarity;
        };

struct neuron {
        double activation;              /* A neuron has an activation
*/
        double newactivation;           /* for calculation purposes
    */
        double sigma;           /* slope constant in sigmoid
function */
        double delta;           /* for backprop
    */
        double tempsigma;       /* for adjusting sigmas
    */
        double neurpole;        /* pole in activation equation  */
        double temppole;        /* for adjustment of poles      */
        double polex1;          /* for dynamics of pole adjustment
*/
        double polex2;          /* also
        */
        double totalinput;      /* the sum of weights *
Simoid(inputs) */
        struct axons axon;      /* and a bunch of axons.        */
        };

typedef struct Layer_struct {
        int neuronsize; /* size in bytes of neurons in this layer */
        int len;        /* number of neurons in the layer
    */
        char inhibition_flag; /* whether of not there is lat. inhib.*/
        char normalize_flag; /* flag for normalizing outputs of layer */
        void *data;             /* The weights and connections
    */

        } Layers;


typedef struct tagTrainingParams {
        long nHoldConst;
        long nIterations;
        double gamma;
        double SigmaAdjustmentFactor;
        double PoleAdjustmentFactor;
        double InputVariation;
        double InputBase;
        double InputBaseVariation;
        long SaveResponseRate;
        long BackupRate;
        long WeightUpdateRate;
        long SigmaUpdateRate;
        long PoleUpdateRate;
        long ActivationResetRate;
        int TrainMode;
        int nOutputLayer;
        int nSaveResponseLayer;
        } TRAININGPARAMS;

/* ------------------------------------------------------------------- */

Layers *Layer;

double sampletime=.005; /* Numerical integration time constant. */


int NumLayers,MaxClassNumber;

TRAININGPARAMS tpTrain;

NETCLASS *lpNetClass;

void BackPropProc();

/* ------------------------------------------------------------------- */

void AssignSizes()

/* define the size of neurons and number of neurons in each  Layer */
/* Note that a model should be loaded before calling this.
        */
{

    int i,size;
    INPUTCLASS * tInp;

    if ((NumLayers=MaxClassNumber)==0)
        {
        printf ("Error -no model loaded, AssignSizes()");
        return;
        }
    Layer = (Layers *) calloc(NumLayers,sizeof(Layers));

    for (i=0;i<NumLayers;++i)
        {
        Layer[i].len=lpNetClass[i].nClassSize;
        for (tInp=&(lpNetClass[i].Input[0]),size=0; tInp-
>wInputClassNum!=EOINPUT;++tInp)
            size += tInp->wConnectionWidth;
        Layer[i].neuronsize=sizeof(struct neuron) +
(size)*sizeof(struct axons);
        }

}

void AssignMem()

/* allocates memory for each Layer's data. */
{
  int i;

    for (i=0;i<NumLayers;++i)
        Layer[i].data=(void *)
calloc(Layer[i].len,Layer[i].neuronsize);

    return;

}
```

```
int FindStartPoint(nWidth,nInp, nOut,NeuronNum)
int nWidth,nInp,nOut,NeuronNum;
{
      int nStart;
      int nLastStart;
      double ForEach;

      if (NeuronNum==0) return 0;
      nLastStart=nOut-nWidth;

      --nInp;
      if (NeuronNum==nInp) return nLastStart;

      ForEach =  (double)  nLastStart / ((double)  nInp);

      return (int) ( (double )  NeuronNum * ForEach +.5);
}

double InitWeight(ConnectionType)
int ConnectionType;
{
      double ran;

      if (ConnectionType&INITIALLYZERO)
            return 0;

      ran = (double)  rand() / (double)  RAND_MAX;

      if (ConnectionType & WEIGHTPOSITIVE) return ran;
      if (ConnectionType & WEIGHTNEGATIVE) return -ran;

      return 2*(ran - .5);
}

int WeightPolarity( ConnectionType)
int ConnectionType;
{
      if (ConnectionType & WEIGHTPOSITIVE) return
WEIGHT_POS;
      if (ConnectionType & WEIGHTNEGATIVE) return
WEIGHT_NEG;

      return WEIGHT_ANY;
}

void DefineStructure()
/* Define the structure of the net by initializing connections. Also sets all
   neural activations to 0 and sigmas to SLOPECONST*/

{
       int i,j,k,nInput;
       long int temp;
   struct neuron *nptr;
   struct axons *axptr;
       NETCLASS  * tNetClass;
       INPUTCLASS  *tInp;
       int nWidth,nSize,nStart,nEnd;


     for (i=0;i<NumLayers;++i)
          {
          tNetClass=&(lpNetClass[i]);
          nSize=Layer[i].len;
          for (j=0;j<nSize;++j)
               {
               nptr=neur(i,j);
               nptr->sigma = SLOPECONST;
               nptr->neurpole = NEURPOLE * (double) rand() /
                              (double) RAND_MAX;
               axptr=&(nptr->axon);
               for (tInp=&(tNetClass->Input[0]); (nInput=tInp-
>wInputClassNum)!= EOINPUT; ++tInp)
                    {
                    if (tInp->ConnectionType &
CONNECTCOHERENT)
                         {
                              nEnd = (nStart=FindStartPoint(tInp-
>wConnectionWidth,nSize,
                              Layer[nInput].len,j)) + tInp-
>wConnectionWidth-1;
                         for (k=nStart;k<=nEnd;++k)
                              {
                              axptr->presynaptic=neur(nInput,k);
                              axptr->weight=InitWeight(tInp-
>ConnectionType);
                              axptr->polarity=WeightPolarity(tInp-
>ConnectionType);
                              axptr->maxinit = 1.0;
                              ++axptr;
                              }
                         }
                    else  /* Connect Random: */
                         {
                         for (k=0;k<tInp->wConnectionWidth;++k)
                              {
                              axptr-
>presynaptic=neur(nInput,rand()%Layer[nInput].len);
                              axptr->weight=InitWeight(tInp-
>ConnectionType);
                              axptr->polarity=WeightPolarity(tInp-
>ConnectionType);
                              axptr->maxinit=1.0;
                              ++axptr;
                              }
                         }
                    }
               }
       axptr->presynaptic=EOL;

       Layer[i].inhibition_flag=FALSE;
       Layer[i].normalize_flag=FALSE;
       }

}

SetupScheduler()

/* sets up weights & thresholds of scheduler layer. */
/* nptr->neurpole is the 'mean', nptr->sigma is sigma. */

{

      int i,n;
      double DivSize,DivStart;
      struct neuron *nptr;
      struct axons * axptr;

      if (SCHEDULER_LAYER >= NumLayers) return;

      n=Layer[SCHEDULER_LAYER].len;
      DivSize = MAXINPUT / (double) n * 1.5 ;
      for (i=0;i<n;++i)
           {
           DivStart = (double) i /  (double) (n-1) * (MAXINPUT -
DivSize);
           nptr= neur(SCHEDULER_LAYER,i);
           nptr->neurpole= DivStart+DivSize/2.0;
           nptr->sigma = DivSize/2.2;
           for (axptr = &(nptr->axon); (axptr->presynaptic)!=EOL;
++axptr)
                axptr->weight  = 1.0;
           }
}


LoadModel()
{
     char cFileName[40];
     int i;

     FILE * fin;
     NETCLASS * tNetClass;
     INPUTCLASS *tInput;
```

```
    sprintf (cFileName,"data/ident.net");
    fin=fopen(cFileName,"r");
    fscanf(fin,"%d",&MaxClassNumber);
    printf ("Loading %d Classes...\n",MaxClassNumber);

    lpNetClass = (NETCLASS *)
calloc(MaxClassNumber,sizeof(NETCLASS));
    for (i=0;i<MaxClassNumber;++i)
        {
        fscanf(fin,"%s",lpNetClass[i].szClassName);
        fscanf(fin,"%d",&(lpNetClass[i].nClassSize));
        tInput = &(lpNetClass[i].Input[0]);
        fscanf(fin,"%d",&(tInput->wInputClassNum));
        while (tInput->wInputClassNum !=EOINPUT)
            {
            fscanf(fin,"%d",&(tInput->ConnectionType));
            fscanf(fin,"%d",&(tInput->wConnectionWidth));
            ++tInput;
            fscanf(fin,"%d",&(tInput->wInputClassNum));
            }
        }

    fclose(fin);



}

void Initialize()
{

     LoadModel();
     AssignSizes();

     AssignMem();
     DefineStructure();
     SetupScheduler();

}

void ShowConnectivity()
/* This routine prints out which neurons are connected to which others.
  It overwrites all neuron's activation & newactivation values. */

{
    int i,j,k;
    long int temp;
    char s[20];
    struct neuron  *nptr;
    struct axons *axptr;

    FILE * f;

    f=fopen("data/showconnections.dat","w");
    for (i=0;i<NumLayers;++i)
        {

        for (j=0;j<Layer[i].len;++j)
            {
            neur(i,j)->newactivation = j;
            neur(i,j)->activation =i;
            }
        }
    for (i=0;i<NumLayers;++i)
        {
        fprintf (f,"Layer %d: %s\n",i,lpNetClass[i].szClassName);
        for (j=0; j< Layer[i].len; ++j)
          {
          fprintf (f,"  Neuron %d\n",j);
          nptr = neur(i,j); axptr=&(nptr->axon);
          while ( (axptr->presynaptic)  != EOL )
            {
            switch (axptr->polarity)
                {
                case WEIGHT_NEG:
                    sprintf(s,"Inhibitory");
                    break;
                case WEIGHT_POS:
                    sprintf (s,"Excitatory");
                    break;
                default:
                    sprintf(s,"Bipolar");
                    break;
                }

            fprintf (f,"     Input is %s, neuron %.0f (%s)\n",
                lpNetClass[(int)(axptr->presynaptic)-
>activation].szClassName,
                (axptr->presynaptic)->newactivation,s);
            ++axptr;
            }
          }
        }
    fclose(f);
}


void SaveWeights(Layernum)

int Layernum;

/* save weights for Layernum */
{
    struct neuron *nptr;
    struct axons *axptr;
    int i;
    char s[20], ans[5];
    FILE *f;

    sprintf (s,"data/wate%d.dat",Layernum);
    printf ("Saving weights - Layer %d\n",Layernum);
    if (DRYRUN) /* override for autosave in long training run */
        if ((f=fopen(s,"r"))!=NULL)
            {
            printf ("Warning -- file %s already exists -- overwrite
?",s);
            scanf ("%s",ans);
            fclose(f);
            if (*ans != 'y')
                return;
            }
    f=fopen(s,"w");
    for (i=0;i<Layer[Layernum].len;++i)
        {
        nptr=neur(Layernum,i);
        fprintf(f,"%.17lf\n",nptr->sigma);
        fprintf(f,"%.17lf\n",nptr->neurpole);
        axptr=&(nptr->axon);
        while ( (axptr->presynaptic) != EOL)
            {
            fprintf(f,"%.17lf\n",axptr->weight);
            ++axptr;
            }
        fprintf (f,"\n");
        }
    fclose(f);
}

void RecoverWeights(Layernum)

int Layernum;

/* recover weights for Layernum */
{
    struct neuron *nptr;
    struct axons *axptr;
    int i;
    char s[20];
    FILE *f;

    sprintf (s,"data/wate%d.dat",Layernum);
    f=fopen(s,"r");
    if (f==NULL)
        {printf ("Error - FNFF - File %s not found.\n",s);
        return;
        }

    for (i=0;i<Layer[Layernum].len;++i)
```

```
        {
        nptr=neur(Layernum,i);
        fscanf(f,"%lf\n",&(nptr->sigma));
        fscanf(f,"%lf\n",&(nptr->neurpole));
        axptr=&(nptr->axon);
        while ( (axptr->presynaptic) != EOL)
            {
            fscanf(f,"%lf\n",&(axptr->weight));
            ++axptr;
            }
        }
    fclose(f);
}

void setactivations(Layernum)
int Layernum;

/* sets activations in Layernum to newactivations in Layernum */
{
    int i,n;
    struct neuron *nptr;

    n=Layer[Layernum].len;
    for (i=0;i<n;++i)
        {
        nptr=neur(Layernum,i);
        nptr->activation = nptr->newactivation;
        }
}


double Sigmoid( t,sigma)
double t, sigma;
/* Non-linear sigmoidal function */
{
    double x;

/*  return  ((t<0) ? 0 : t*sigma);  */

    return  ((t<0) ? 0 : 1/(1+exp(-sigma*t))-.5);
/*  return  1/(1+exp(-sigma*t))-.5;     */

}

double SigmoidDeriv(t,sigma)
double t,sigma;
/* Derivative of above function */
{
    register double x;

    if (t<0) return 0;
/*  return sigma;   */

    x=Sigmoid(t,sigma);
    return sigma*(.25-x*x); /* (a^2-b^2)=(a+b)(a-b) */


}


void normalizeLayer(Layernum)
int Layernum;

/* Normalizes vector of activations in Layer Layernum to unity    */
{
    int i,n;
    struct neuron *nptr;
    double sum,*t;

    n=Layer[Layernum].len;
    sum=0;
    for (i=0;i<n;++i)
        {
        t=&(neur(Layernum,i)->activation);
        sum += *t * *t;
        }
    sum=sqrt(sum);
    for (i=0;i<n;++i)
        {
        t=&(neur(Layernum,i)->activation);
        *t = *t/sum;
        }
}

void CalcScheduler()
/* Calculates scheduler's activations.  It's inputs are assumed calculated.
*/
{
    int j,n;
    double  t,mid,k;
    struct neuron * nptr, *presynptr;
    struct axons *axptr;
    Layers *tLayer;

    tLayer = &(Layer[SCHEDULER_LAYER]);
    n=tLayer->len; t=0;
    for (j=0;j<n;++j)
        {
        nptr=neur(SCHEDULER_LAYER,j);
        t=0;
        axptr=&(nptr->axon);
        while ( (presynptr=(axptr->presynaptic))!=EOL)
            {
            t += axptr->weight * presynptr->activation;
            ++axptr;
            }

        k=(t-nptr->neurpole)/nptr->sigma;

        nptr->newactivation=.1*(1.0-exp(-k*k));
/*
    Parabola, max of 1.0 :

        mid = (nptr->neurpole+nptr->sigma)/2.0;
        k = 4.0/((nptr->neurpole-nptr->sigma)*
            (nptr->neurpole-nptr->sigma));
        t = k*(t-mid)*(t-mid);
        if (t>1.0) t=1.0;
        nptr->newactivation=t;

    if ( (t<nptr->neurpole) || (t>nptr->sigma))
        nptr->newactivation = 1.0*mid*mid;
    else
        nptr->newactivation = 1.0*(t-mid)*(t-mid);

    ----------------------------

    Hard limiting threshold :

        if (t > nptr->sigma) t = -1e10;
        nptr->newactivation = ( (t < nptr->neurpole) ? 0 : 1 );   */

        }
    setactivations(SCHEDULER_LAYER);

}


int CalcActivations(Layernum)

int Layernum;

/* Calculates Layernum's activations.  It's inputs are assumed calculated.
    It returns 0. */
{
    int j,k,n;
    double  t,temp,sigma;
    struct neuron * nptr, *presynptr;
    struct axons *axptr;
    Layers *tLayer;

    if (Layernum==SCHEDULER_LAYER)
        {
        CalcScheduler();
```

```
            return;
            }

      tLayer = &(Layer[Layernum]);
      n=tLayer->len; t=0;
      for (j=0;j<n;++j)
            {
            nptr=neur(Layernum,j);
            t=0;
            sigma=nptr->sigma;
            axptr=&(nptr->axon);
            while ( (presynptr=(axptr->presynaptic))!=EOL)
                  {
                  t += axptr->weight * Sigmoid(presynptr-
>activation,sigma);
                  ++axptr;
                  }
            nptr->totalinput = t;
            nptr->newactivation = (1-nptr->neurpole*sampletime)*nptr-
>activation + sampletime*t;
            }
      setactivations(Layernum);
      return 0;

}


SaveAllWeights()
{
      int i;

      for (i=0;i<NumLayers;++i)
            SaveWeights(i);
}

RecoverAllWeights()
{
      int i;

      for (i=0;i<NumLayers;++i)
            RecoverWeights(i);
}



void reset()

/* resets activations to 0 */
{
      int i,j,n;

      for (i=0;i<NumLayers;++i)
            {
            n=Layer[i].len;
            for (j=0;j<n;++j)
                  neur(i,j)->activation=0.0;
            }
}

void UpdatePoles()
/* adds values in nptr->temppole to nptr->pole for all neurons,
   resets temppoles to 0. */
{
   int i,j,n;
   register struct neuron *nptr;

   for (i=0;i<NumLayers;++i)
     if (i!=SCHEDULER_LAYER)
       {
       n=Layer[i].len;
       for (j=0;j<n;++j)
         {
         nptr=neur(i,j);

         nptr->neurpole += nptr ->temppole;
         nptr->temppole = 0;
         if (nptr->neurpole < 0.01)
               nptr->neurpole=0.01;
         }
      }

}

void UpdateSigmas()
/* adds values in nptr->tempsigma to axptr->sigma for all neurons,
   resets tempsigmas to 0. */
{
   int i,j,n;
   register struct neuron *nptr;

   for (i=0;i<NumLayers;++i)
      {
      n=Layer[i].len;
      for (j=0;j<n;++j)
        {
        nptr=neur(i,j);

        nptr->sigma += nptr ->tempsigma;
        if (nptr->sigma < .001)
              nptr->sigma=.001;
        }
      }

}


void UpdateWeights()

/* adds value Rate*axptr->tempweight to axptr->weight for all weights,
   resets tempweight to 0. */
{
   int i,j,n;
      register struct axons *axptr;
   struct neuron *nptr;

   for (i=0;i<NumLayers;++i)
      {
      n=Layer[i].len;
      for (j=0;j<n;++j)
        {
        nptr=neur(i,j);
        axptr=&(nptr->axon);

        while(axptr->presynaptic != EOL)
          {
          axptr->weight += axptr->tempweight;

          if (axptr->polarity==WEIGHT_POS)
              axptr->weight=max(0.0,axptr->weight);
          else if (axptr->polarity==WEIGHT_NEG)
              axptr->weight=min(0.0,axptr->weight);

          axptr->tempweight=0;
          ++axptr;
          }
        }
      }

}


void InitializeWeights()
/* Initializes Weights based on value of axptr->polarity
      */
{
      int i,j,n;
      register struct axons * axptr;
      struct neuron * presynptr;
      double  ran;


      for (i=0;i<NumLayers;++i)
        if (i!=SCHEDULER_LAYER)     /* Scheduler layer is already
set up in SetupScheduler() */
            {
            n=Layer[i].len;
            for (j=0;j<n;++j)
                  {
```

```
                for (axptr=&(neur(i,j)->axon); (presynptr=axptr->presynaptic)!=EOL;++axptr)
                    {
                    ran=(double ) rand() / (double ) RAND_MAX;
                    switch (axptr->polarity)
                        {
                        case WEIGHT_NEG:
                            ran = -ran; break;
                        case WEIGHT_ANY:
                            ran = 1-2*ran; break;
                        default : break;
                        }
                    axptr->weight = ran * axptr->maxinit;
                    }
                }
            }
}

double ModelResponse( y, inp)
double y,inp;
{
   double t;

      t= sampletime*(inp-(y)*(1+4.0*y*y));
   return y+t;
}


void LoadTrainParams()
{
    FILE * f;
    char szFieldName[50];

    if ( (f=fopen("data/tp.dat","r"))==NULL)
        {
        printf ("Error - data/tp.dat not found\n");
        return ;
        }

    while ( fscanf (f,"%s",szFieldName) != EOF )
        {
        if ( !strcmp(szFieldName,"HoldConst:"))
            fscanf (f,"%ld",&(tpTrain.nHoldConst));

        else if ( !strcmp(szFieldName,"Iterations:"))
            fscanf (f,"%ld",&(tpTrain.nIterations));

        else if ( !strcmp(szFieldName,"PoleAdjustmentFactor:"))
            fscanf (f,"%lf",&(tpTrain.PoleAdjustmentFactor));

        else if ( !strcmp(szFieldName,"SigmaAdjustmentFactor:"))
            fscanf (f,"%lf",&(tpTrain.SigmaAdjustmentFactor));

        else if ( !strcmp(szFieldName,"Gamma:"))
            fscanf (f,"%lf",&(tpTrain.gamma));

        else if ( !strcmp(szFieldName,"InputVariation:"))
            fscanf (f,"%lf",&(tpTrain.InputVariation));

        else if ( !strcmp(szFieldName,"InputBase:"))
            fscanf (f,"%lf",&(tpTrain.InputBase));

        else if ( !strcmp(szFieldName,"InputBaseVariation:"))
            fscanf (f,"%lf",&(tpTrain.InputBaseVariation));

        else if ( !strcmp(szFieldName,"SaveResponseRate:"))
            fscanf (f,"%ld",&(tpTrain.SaveResponseRate));

        else if ( !strcmp(szFieldName,"BackupRate:"))
            fscanf (f,"%ld",&(tpTrain.BackupRate));

        else if ( !strcmp(szFieldName,"PoleUpdateRate:"))
            fscanf (f,"%ld",&(tpTrain.PoleUpdateRate));

        else if ( !strcmp(szFieldName,"SigmaUpdateRate:"))
            fscanf (f,"%ld",&(tpTrain.SigmaUpdateRate));

        else if ( !strcmp(szFieldName,"WeightUpdateRate:"))
            fscanf (f,"%ld",&(tpTrain.WeightUpdateRate));

        else if ( !strcmp(szFieldName,"TrainMode:"))
            fscanf (f,"%d",&(tpTrain.TrainMode));

        else if ( !strcmp(szFieldName,"OutputLayer:"))
            fscanf (f,"%d",&(tpTrain.nOutputLayer));

        else if ( !strcmp(szFieldName,"SaveResponseLayer:"))
            fscanf (f,"%d",&(tpTrain.nSaveResponseLayer));

        else if ( !strcmp(szFieldName,"ActivationResetRate:"))
            fscanf (f,"%ld",&(tpTrain.ActivationResetRate));


        }


    printf("hold const: %ld\n",tpTrain.nHoldConst);
    printf("Iterations: %ld\n",tpTrain.nIterations);
    printf("Gamma: %lf\n",tpTrain.gamma);
    printf("SigmaAdjustmentFactor:
%lf\n",tpTrain.SigmaAdjustmentFactor);
    printf("PoleAdjustmentFactor:
%lf\n",tpTrain.PoleAdjustmentFactor);
    printf("Base Input : %lf\n",tpTrain.InputBase);
    printf("Inpu DC Variation : %lf\n",tpTrain.InputBaseVariation);
    printf("Input Variation: %lf\n",tpTrain.InputVariation);
    printf("SaveResponseRate: %ld\n",tpTrain.SaveResponseRate);
    printf("BackupRate: %ld\n",tpTrain.BackupRate);
    printf("WeightUpdateRate: %ld\n",tpTrain.WeightUpdateRate);
    printf("SigmaUpdateRate: %ld\n",tpTrain.SigmaUpdateRate);
    printf("PoleUpdateRate: %ld\n",tpTrain.PoleUpdateRate);
    printf("ActivationResetRate: %ld\n",tpTrain.ActivationResetRate);
    printf("OutputLayer: %d\nSaveResponseLayer: %d\n",
        tpTrain.nOutputLayer,tpTrain.nSaveResponseLayer);
    printf("TrainMode: %s\n",(tpTrain.TrainMode==SINEINPUT ?
"Sine Wave" :
                    (tpTrain.TrainMode==STEPINPUT ? "Step" :
                     "Noise" )));
    fclose(f);
}

void train(mode)
int mode;

/* Trains the network weights. */
{
    int filestatus;
    char ans;

    switch (mode) {
        case 1:
            InitializeWeights();
            printf ("Starting backprop :\n");
            BackPropProc();
            printf ("Well, I'm glad that's over\n");
            SaveAllWeights();
            break;
        case 2:
            RecoverAllWeights();
            BackPropProc();
            SaveAllWeights();
            break;
        case 3: case 4:
            RecoverAllWeights();
            break;
            }
}

void ResetActivations()

/* resets activations to 0 */
{
   int i,j,n;

   for (i=0;i<NumLayers;++i)
      {
      n=Layer[i].len;
```

```
        for (j=0;j<n;++j)
            neur(i,j)->activation=0.0;
        }
}

void RunMode()

{
    int i,k=0,n1,j,printout_rate,n2,mode;
    double  input=1,target=0,sum=0,err,t,y1=0,Base,facter;
    FILE *f;
    FILE *fduh;


    LoadTrainParams();
    f=fopen("duh.dat","w");
    fduh=fopen("duh1.dat","w");
    n2=2*tpTrain.nHoldConst;
    n1=max(n2/100,1);
    if (tpTrain.TrainMode==SINEINPUT)
        facter=2.0*PI/(double) n2;
    else if (tpTrain.TrainMode==NOISEINPUT)
        facter = tpTrain.InputVariation / (double) RAND_MAX;

    Base = tpTrain.InputBase + tpTrain.InputBaseVariation *
        (double) rand() / (double) RAND_MAX  ;


    ResetActivations();

    input=(neur(0,0)->activation=Base);
/*  for (k=0;k<n2;++k)
        {
        y1=ModelResponse(y1,input);
        for (j=1;j<NumLayers;++j)
            {
            CalcActivations(j);
            }
        }
*/

    if (tpTrain.TrainMode==STEPINPUT)
        input=neur(0,0)->activation= tpTrain.InputBase +
         (double)rand()/(double)RAND_MAX *
tpTrain.InputBaseVariation;

    for (k=0;k<n2;++k)
        {

        if (tpTrain.TrainMode==SINEINPUT)
            {
            input=1.0+sin(facter*(double ) k );

            input *= tpTrain.InputVariation;
            input += Base;
            }
        else if (tpTrain.TrainMode==STEPINPUT)
            {
/*          if (k >(n2/2))
                input=Base;
*/
            }

        else {
            input = facter* (double) rand() ;
            input += Base;
           }

        y1=ModelResponse(y1,input);
        target=y1;

        neur(0,0)->activation=input;

        for (j=1;j<NumLayers;++j)
            {
            CalcActivations(j);
            }
        err=neur(tpTrain.nOutputLayer,0)->activation - target;
        if ( !(k%n1))
            {
            t=sampletime*((double )n1*(double )(k+1));
            fprintf(f,"%lf\t%lf\t%lf\t%lf\n",t,target,
                neur(tpTrain.nOutputLayer,0)->activation,input);

            for (j=0;j<Layer[tpTrain.nSaveResponseLayer].len;++j)
             fprintf(fduh,"%lf\t",
               neur(tpTrain.nSaveResponseLayer,j)->activation);
            fprintf(fduh,"\n");

            }
        }
}

void PrintStuff()

{
    int i,n;
    FILE *f;

    n = Layer[2].len;
    f=fopen("duh2.dat","w");

    for (i=0;i<n;++i)
        {
        fprintf(f,"%17lf\n",neur(2,i)->neurpole);
        }
    fclose(f);
}

void main()
{
    int seed,mode,i;
    char c[10];

    printf ("Hello, world.\n");
    printf ("Now enter the random seed. ");
    scanf ("%d",&seed);
    printf ("Thank you.\n\n");
    printf ("Yo, enter the type of run :\n\t1) Train (first time)\n\t2)
Load old weights and train some more\n");
    printf ("\t3) Run mode\n");
    printf ("\t4) Print stuff\n\n");
    query ("Which ->","%d",&mode);
    srand(seed);
    Initialize();


    ShowConnectivity();


    printf ("Calling train....\n");
    train(mode);
    printf ("Back already\n");
    if (mode == 3)
        RunMode();
    if (mode == 4)
        PrintStuff();

    free(lpNetClass);
    for (i=0;i<NumLayers;++i)
        free(Layer[i].data);
    free(Layer);

}




void SetOutputDeltas( fDesired, nLayer)

double  * fDesired;
int nLayer;
/* Sets the Deltas in Layer nLayer (the output) to the values in vector
fDesired. */
```

```
{
    int i,n;
    register struct neuron * nptr;

    n=Layer[nLayer].len;
    for (i=0;i<n;++i)
        {
        nptr=neur(nLayer,i);
        nptr->delta = nptr->activation - *(fDesired++);
        }

}

void ResetDeltas(bResetAxonStuff)
BOOL bResetAxonStuff;

/** Sets the Deltas in all Layers to 0. Also resets all axptr->tempdelta
    and axtpr->temppsi values to 0 if bResetAxonStuff =
DERIVRESET. */

{
    int i,j,n;
    register struct axons * axptr;
    register struct neuron * nptr;

    for (i=0;i<NumLayers;++i)
        {
        n=Layer[i].len;
        for (j=0;j<n;++j)
          {
          (nptr=neur(i,j))->delta = 0;
          nptr->polex1=nptr->polex2=0;
          if (bResetAxonStuff==DERIVRESET)
           {
           for((axptr=&(neur(i,j)->axon));(axptr-
>presynaptic)!=EOL;
                ++axptr)
                    {
                    axptr->tempdelta=0;
                    axptr->temppsi=0;
                    }
          }
        }

}

void CalcDeltas(nLayer)

int nLayer;

/* Assumes that the deltas for this Layer are already set; propagates the
    error back to its inputs. It adds this contribution to the deltas of
    its inputs.
    In this program, the deltas are calculated taking into account the
    effect of the neurons built-in dynamics. */


{
    register struct axons * axptr;
    register struct neuron * presynptr;
    struct neuron * nptr;
    double  deltak,sigma,pole;
    int j,n;

    n=Layer[nLayer].len;
    for (j=0;j<n;++j)
        {
        nptr=neur(nLayer,j);
        sigma=nptr->sigma;
        pole=nptr->neurpole;
        deltak = nptr->delta;
        for (axptr=&(nptr->axon); (presynptr=axptr-
>presynaptic)!=EOL; ++ axptr)
            {
            axptr->tempdelta+=-pole*sampletime*axptr->tempdelta
+
                sampletime*axptr->weight*
                SigmoidDeriv(presynptr->activation,sigma);
                presynptr->delta += deltak * axptr->tempdelta;
                }
        }
}


void ShowDeltas(nLayer)
int nLayer;
{
    int i,n;

    n=Layer[nLayer].len;
    for (i=0;i<n;++i)
        printf("(%.3lg, %.3lg)\t",neur(nLayer,i)->delta,neur(nLayer,i)-
>activation);
    printf ("\n");
}

int BackPropPath1[]= { 3,-1 };
int StaticLayer = 1000;

void AdaptPoles(Rate)

double Rate; /* Unused */
/* updates all temppole variables in all neurons. does not call
UpdatePoles. */

{
    int i,n,j;
    double sum,sigma,x;
    register struct axons * axptr;
    struct neuron * presynptr, * nptr;

    for (i=0;i<NumLayers;++i)
      if ((i!=StaticLayer)&&(i!=SCHEDULER_LAYER))
        {
        n=Layer[i].len;
        for (j=0;j<n;++j)
            {
            nptr=neur(i,j);
            x = sampletime*nptr->polex2;
            nptr->polex2 += -sampletime*(nptr->polex1+
                    2*nptr->neurpole*nptr->polex2 +
                    nptr->totalinput);


            nptr->temppole -= (nptr->polex1 += x)*
                        nptr->delta;
            }
        }



}

void AdaptSigmas(Rate)

double Rate;
/* updates all tempsigma variables in all neurons. does not call
UpdateSigmas. */

{
    int i,n,j;
    double sum,sigma,x;
    register struct axons * axptr;
    struct neuron * presynptr, * nptr;

    for (i=0;i<NumLayers ;++i)
      if ((i!=StaticLayer)&&(i!=SCHEDULER_LAYER))
        {
        n=Layer[i].len;
        for (j=0;j<n;++j)
            {
            sum=0;
            nptr=neur(i,j);
            sigma=nptr->sigma;
            for (axptr=&((nptr=neur(i,j))->axon); (presynptr=axptr-
>presynaptic)!=EOL;++axptr)
                {
```

```
                    x=Sigmoid(presynptr->activation,sigma);
                    sum += axptr->weight*presynptr->activation*(.25-
x*x);
                    }
                nptr->tempsigma -= Rate*nptr->delta*sum;
                }
            }


}

void BackProp(gamma)

double gamma;

/* Propagates error along BackPropPath using CalcDeltas.  Then,
updates
 tempweights in ALL layers.  Does not call updateweights().  Also,
 it will NOT adjust the weights of the Scheduler layer.  */

{
    int i,j,n,nLayer;
    register struct axons * axptr;
    struct neuron * presynptr, * nptr;
    char buff[80];



    for (i=0;(nLayer=BackPropPath1[i])!=-1;++i)
        {
        CalcDeltas(nLayer);
        }


    for (i=0;i<NumLayers ;++i)
      if ((i!=SCHEDULER_LAYER) && (i!=StaticLayer))
        {
        n=Layer[i].len;
        for (j=0;j<n;++j)
            {
            for (axptr=&((nptr=neur(i,j))->axon); (presynptr=axptr-
>presynaptic)!=EOL;++axptr)
                {
                axptr->temppsi += -nptr->neurpole*sampletime*
                  axptr->temppsi + sampletime *
                  Sigmoid(presynptr->activation,nptr->sigma);
                axptr->tempweight -= gamma*axptr->temppsi *
                  nptr->delta/nptr->neurpole;
                }
            }
        }
}

void Errout(f, sum)

FILE * f; double  sum;
{
    fprintf (f,"%lf\n",sum);
}




void BackPropProc()

/* Trains the network weights.

*/

{

    int Mode,j,k,duh;

    long i,lItCount=0;
    double  yModel=0,Input,sum,err;
    double  gamma =1e-4,facter,Base;
    char buff[80];
    FILE * f,*fresponse ;
    BOOL writethis;

    f=fopen("errout.dat","w");
    fresponse=fopen("response.dat","w");


    printf ("Beginning Training...\n");
    LoadTrainParams();
    ResetActivations();

    facter=2.0*PI/ (double) tpTrain.nHoldConst;
    for (i=0;i<tpTrain.nIterations;++i)
        {
        sum=0.0;
        if (!(i%tpTrain.ActivationResetRate) )
            {
            yModel=0;
            Base = tpTrain.InputBase + tpTrain.InputBaseVariation*
                (double) rand() / (double) RAND_MAX;
            ResetActivations();
            ResetDeltas(DERIVRESET);
            }

        if ( (!DRYRUN)&& (i>0 &&
((i%tpTrain.BackupRate)==0)))
            SaveAllWeights();

        if ( ((i%tpTrain.SaveResponseRate==0)) ||
           (i%tpTrain.SaveResponseRate==1))
            writethis=TRUE;
        else
            writethis=FALSE;
        if (tpTrain.TrainMode==STEPINPUT)
            {
            Input=neur(0,0)->activation= tpTrain.InputBase +
             (double) rand()/(double) RAND_MAX *
                tpTrain.InputBaseVariation;
            }
        for (k=0;k<tpTrain.nHoldConst;++k)
            {
            ++lItCount;

            switch (tpTrain.TrainMode)
                {
                case SINEINPUT:
                    Input=sin(facter*(double ) k );
                    Input *= tpTrain.InputVariation;
                    neur(0,0)->activation=(Input+=Base);
                    break;

                case NOISEINPUT:
                    Input=(double) rand()/(double)
RAND_MAX;
                    Input *= tpTrain.InputVariation;
                    neur(0,0)->activation=(Input+=Base);
                    break;


                }

            for (j=1;j<NumLayers;++j)

                CalcActivations(j);

            yModel=ModelResponse(yModel,Input);

            ResetDeltas(NODERIVRESET);

            SetOutputDeltas(&yModel,tpTrain.nOutputLayer);

            err = neur(tpTrain.nOutputLayer,0)->activation -
yModel;

            sum += err*err;
```

```
            BackProp(tpTrain.gamma);  /* Propagate error  */

        /*  if ( (rand()%1000)==313)
                {
                duh=rand()%NumLayers;
                printf ("Layer %d Deltas :\n\t",duh);
                ShowDeltas(4);
                }                   */
/*
            AdaptSigmas(tpTrain.SigmaAdjustmentFactor);
*/
            AdaptPoles(tpTrain.PoleAdjustmentFactor);



            if (writethis==TRUE)
             {
             if ((k%10)==0)
              fprintf(fresponse,"%lf\t%lf\n",
                yModel,neur(tpTrain.nOutputLayer,0)->activation);
             }

            if (!(IItCount%tpTrain.WeightUpdateRate))
                {
                UpdateWeights();
                }
            if (!(IItCount%tpTrain.PoleUpdateRate))
                {
                UpdatePoles();
                }
/*
            if (!(IItCount%tpTrain.SigmaUpdateRate))
                {
                UpdateSigmas();
                }
*/


            }
        Errout(f,sum);

        }
    UpdateWeights();
/*  UpdateSigmas();         */
    UpdatePoles();
    fclose(f);
    return;
}
```

Vita

Surname: Jubien | Given Names: Christopher Michael

Date of Birth: March 6, 1967. | Place of Birth: Montreal, Quebec

Educational Institutions Attended:

University of Waterloo	1985 to 1990
University of Victoria	1990 to 1992

Degrees Awarded:

B.A.Sc., University of Waterloo	1990

Honors and Awards:

NSERC PGS	1990-92
University of Victoria President's Research Grant,	1990-92
Graduate Teaching Fellowship	1991-92
NSERC URA	1989-90

Publications

C. Jubien and M. Jernigan, "Neural Networks for Deblurring Images," *IEEE PACRIM Conference on Computers and Communications,* 1991.

C. Jubien and N. Dimopoulos, "Stable Neural Networks for Systems Identification," *International Symposium on Circuits and Systems*, 1993.

C. Jubien and N. Dimopoulos, "Stable Neural Networks for Identification of a PUMA-560 Robot," *International Conference on Neural Networks*, 1993.

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis: Asymptotically Stable Neural Networks for Identification of Nonlinear Dynamic Systems

Author: Chris Jubien

Date: April 1, 1993.