TRAINING GENERAL DYNAMIC NEURAL NETWORKS

By

ORLANDO DE JESUS

Engineer in Electronics
Universidad Simón Bolívar
Caracas, Venezuela
1985

Project Management Specialist
Universidad Simón Bolívar
Caracas, Venezuela
1992

Master of Science
Oklahoma State University
Stillwater, Oklahoma
1998

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
August, 2002

TRAINING GENERAL DYNAMIC NEURAL

NETWORKS

Thesis Approved:

Martin T Hagan

Thesis Adviser

Carl D Lat

Misawa

Timothy J. Pettibone

Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to thank:

Dr. Martin Hagan, my major adviser, for his guidance, encouragement, patience and friendship. The dissertation presented here is the result of many hours of discussion and their headaches trying to understand how the training of dynamic networks works.

Dr. Eduardo Misawa, Dr. Carl Latino and Dr. Gary Yen, my other committee members, for their help and encouragement.

Dr. Howard Demuth of the University of Colorado for his help, support and good sense of humor in the development of this dissertation and its final implementation.

Mark Beale of MHB, Inc. for his help and support in the development of this dissertation and its final implementation.

Roger Schultz of Oklahoma State University and Halliburton Energy Services for his support, encouragement, friendship and incredible exchange of ideas.

Jason Horn of Oklahoma State University for his incredible help in the error surface analysis chapter.

Ohm Arjpolson Pukrittayakamee of Oklahoma State University for his help in the testing of the Neural Network control blocks.

Randall Beckloff and Regina Henry of International Students and Scholars at OSU for their help in all the immigration paperwork and their friendship.

Dr. Dingding Chen and Dr. Meng Fun of Oklahoma State University for their comments, help and support.

Dr. Syed Hamid of Halliburton Energy Services for his support, encouragement and friendship during my work in Carrollton.

Marisela, my lovely wife, for her years of loving support, encouragement and patience. Her infinite support allowed the conclusion of this project.

Our Daughter, that will be coming soon. I hope she will understand the crazy parents she has and the tribulations we are having now.

Manuel and Nelli, my parents, for all they have done to support all my years of study and work, especially the encouragement for this one.

My family Jose, Gilberto, Pedro, Ernesto, Maria, Jorge, Nelli, Juan, Genesis, Elier, Daniela, Debbi, Maria de los Angeles, Belkys, Debora, Elia.

My friends from AETI C.A. The company is gone after many years of incredible results. However the memories will continue forever.

My friends from the Halliburton Energy Services for their help and support.

My friends for your support Aitzol, Xabier and Iñaki, Francis, Carmen Elena and Alberto, Betty and Alfonso, Arleen and Hector, Janice and Eliecer, Carmen and Antonio, Adriana and Alejandro, Francisca and Hanz, Yulissa, Maite and Daniel, Manuel and Ivette, Felipe, Fidelia and Jose Abraham, Yamirca and Jose Luis, Rossany and Noe, Tamar and Victor, Sandra and Gustavo, Elsa and Rene, Paloma and Douglas, Mayra and Gabriel, Claudia and Chicho, Hector, Pascualino, Otto, Magaly and Manuel.

TABLE OF CONTENTS

CHAPTER 1

CHAPTER 2

CHAPTER 3

CHAPTER 4

CHAPTER 5

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

$\mathbf{a}^k$	Neural Network Output for layer k.
ARMA	Autoregressive Moving Average.
$\mathbf{b}$	bias.
BFGS	Broyden, Fletcher, Goldfarb, and Shanno quasi-Newton algorithm.
BTT	Backpropagation Through Time.
CMAC	Cerebellar model articulation controller.
CSTR	Continuous Stirred Tank Reactor [5].
$DI_{m,l}$	Set of all delays in the tapped delay line between Input l and Layer m.
$DL_{m,l}$	Set of all delays in the tapped delay line between Layer l and Layer m.
$E[\ ]$	Expected value.
$E_{LW}^{U}(x)$	Set of all output layer numbers where the layer weight connecting the layer u to the layer x exists.
$E_{LW}^{X}(u)$	Set of all input layer numbers where the layer weight connecting the layer u to the layer x exists.
$E_{S}^{U}(x)$	Set of all output layer numbers where the sensitivity from layer u to layer x exists.
$E_{S}^{X}(u)$	Set of all input layer numbers where the sensitivity from layer u to layer x exists.
$E_{S}(u)$	Set of all layer numbers where the sensitivity from layer u to layer x exists.
$\mathbf{f}^k$	Transfer function for layer k.
FIR	Finite Impulse Response.
fix	Rounds toward zero.
$\dot{\mathbf{F}}^{m}(\mathbf{n}^{m})$	Transfer function derivative for layer m.
FP	Forward Perturbation.

$F(\mathbf{x})$	Performance index for the network.
I	Engine Inertia (*lb-ft-sec*^{2}).
IIR	Infinite Impulse Response.
I_m	Set of indices of input vectors that connect forward to layer *m*.
$\mathbf{IW}^{i,j}$	Input Weight. where j denotes the number of the input vector that enters the weight, and i denotes the number of the layer to which the weight is connected.
K_d	Derivative gain of PID controller.
K_i	Integral gain of PID controller.
K_p	Proportional gain of PID controller.
LDDN	Layered Digital Dynamic Network.
LFFN	Layered Feedforward Network.
L_m^f	Set of indices of layers that directly connect *forward* to layer *m*.
L_m^b	Set of indices of layers that directly connect *backwards* to layer *m*.
$\mathbf{LW}^{i,j}$	Layer Weight, where j denotes the number of the layer coming into the weight and i denotes the number of the layer at the output of the weight.
MagLev	Magnetic Levitation System.
mse	Mean squared error.
msereg	Mean squared error with regularization.
$\mathbf{n}^k$	Input for transfer function of layer *k*.
NARMA	Nonlinear Autoregressive Moving Average.
NN	Neural Network.
$\mathbf{p}^k$	External Neural Network Input *k*.
π	3.14159265358979.
s	Laplace transform variable.
$\mathbf{s}^m$	Sensitivity of layer *m*.
$\mathbf{S}^{u,m}(t)$	Sensitivity calculated from the input of layer *m* to the output of layer *u*.
T	Sampling time.

$\mathbf{t}^k$ Neural Network Target k.

TDL Tapped Delay Line.

U Set of all output layer numbers.

vec Operator that transforms a matrix into a vector by stacking the columns of the matrix one underneath the other [29].

$\mathbf{x}$ Vector containing all of the weights and biases in the network.

X Set of all input layer numbers.

$\mathbf{y}^k$ Neural Network Output k.

z Z-transform variable.

CHAPTER 1

INTRODUCTION

The principal objective of this research is to develop gradient-based algorithms for training general dynamic neural networks. There are three main steps in this development.

The first step is to define a general framework that can be used to represent a large class of dynamic neural networks. Chapter 2 describes such a framework: the Layered Digital Dynamic Network (LDDN). We show that many popular dynamic networks can be represented by the LDDN framework.

The second step is to derive general procedures for computing the gradients for the LDDN. The basic concepts that are used for gradient-based optimization are introduced in Chapter 3. Then Chapters 4 and 5 present the key results of this research: two different algorithms for computing gradients and two different algorithms for computing Jacobians for the LDDN. Chapter 4 develops the general Forward Perturbation (FP) algorithms, and Chapter 5 develops the general backpropagation-through-time (BTT) algorithms.

The third step is the development of gradient-based optimization algorithms that are well suited for dynamic network training. Chapter 6 analyzes the error surfaces for some simple dynamic networks and presents a newly discovered feature of these error surfaces: spurious narrow valleys that can trap optimization algorithms. Based on the analysis of these spurious valleys, Chapter 6 presents modified training algorithms that show improved performance on dynamic networks.

Chapter 7 describes two applications of dynamic network training related to Model Reference Control and Nonlinear Filtering. The control application is solved using the FP algorithm. The nonlinear filtering application is solved using the BTT algorithm.

Chapter 8 presents memory, speed and computational complexity comparisons for the FP and BTT algorithms using 33 different neural networks. Some of these neural networks came from the literature, and others were created for testing purposes.

Finally, Chapter 9 provides a summary of the key results and some directions for future research.

Chapter 2

LAYERED DIGITAL DYNAMIC NEURAL NETWORK

Neural networks can be classified into dynamic and static categories. Static (feedforward) networks have no feedback elements and contain no delays; the output is calculated directly from the input through feedforward connections. In dynamic networks the output depends not only on the current input to the network, but also on the current or previous inputs, outputs or states of the network. This includes feedforward neural networks with delays between layers. In this research we are concerned with the training of general dynamic networks.

In order to develop training algorithms that will be suitable for a general class of dynamic network, we first need to find a general framework. In this chapter we introduce the Layered Digital Dynamic Network (LDDN). We will show how this framework is able to encompass many previously published dynamic architectures.

2.1. Neural Networks.

Biological neural networks allow the interaction between you and me. Not only do they facilitate my writing skills throughout this document and your ability to understand my ideas but they also allow important neurological functions. Each one of us have about 10^{11} highly interconnected neurons, where each interaction is based on tissue, chemical and electrical connections. The biological neural networks inspired simulation using artificial models. This document will consider a class of neural networks called *artificial dynamic neural networks*.

Early work on Neural Networks occurred between late 19th and early 20th centuries. Theories of learning, vision, conditioning, etc. were developed by Hermann von Helmholtz, Ernst Mach, Ivan Parlov and others. McCulloch and Pitts [30] showed that artificial neural networks could compute any arithmetic or logical function in the early 1940s. Later, Hebb [22] proposed the first learning rules based on neuron behavior at the cellular level. Rosenblatt [41] proposed the perceptron network and its associated learning rule in the late 1950s. Unfortunately, the perceptron was only able to solve a limited class of problems. In 1960 the Widrow-Hoff learning rule was introduced to train adaptive linear neural networks [52]. The previous neural networks were very limited in their implementations and applications. Those limitations were presented in the Minsky and Papert book [32], resulting in a slow down in neural network research, except for work due to Kohonen [25] and Anderson [1] on neural networks as memories, and Grossberg [18] on self-organizing networks.

Two developments precipitated an explosion in research involving theoretical and practical applications of neural networks. The first was the introduction of Hopfield networks [23], a class of recurrent neural networks that can be used as associative memories. The second was the development of the backpropagation algorithm [42] for training multilayer networks. Two important results related to the training of recurrent neural networks are the Backpropagation Through Time algorithm [51] and the Forward Perturbation algorithm [54]. These two algorithms will be discussed in detail in later chapters.

2.2. Dynamic Neural Network Architectures.

We will begin this section by reviewing a number of dynamic network architectures that have been proposed in the literature. Then we will present a general class of dynamic network that includes most previous networks as subclasses. This general class of network (called the Layered Digital Dynamic Network) will enable us to develop general purpose training algorithms for dynamic networks.

Tsoi [48] described several architectures enabling the configuration of recurrent neural networks and explained how each architecture helps the gradient evaluation. The Williams-Zipser architecture [54] classifies neurons as input neurons, output neurons, hidden neurons or a combination of input and output. The connection between neurons is an adjustable time delay. The Time Delay Neural Network [49] is a multilayer feedforward neural network where tap delay lines can be connected to the outputs of input neurons, hidden neurons and/or output neurons. The Canonical Form Network [26, 36] consists of a feedforward neural network where the outputs are delayed using a tap delay line with *d* delays as shown in Figure 2.1.

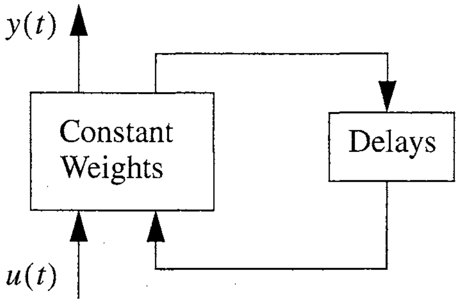

Figure 2.1: *Canonical form networks as introduced by Nerrand et al.*[36]

Narendra et al. [35, 34] proposed six different architectures based on the autoregressive moving average (ARMA) model [4]. The first four models are described by the following nonlinear difference equations [35]:

Model I:

$$y_p(k+1) = \sum_{i=0}^{n-1} \alpha_i y_p(k-i) + g[u(k), u(k-1), \ldots, u(k-m+1)] \tag{2.1}$$

Model II:

$$y_p(k+1) = f[y_p(k), y_p(k-1), \ldots, y_p(k-n+1)] + \sum_{i=0}^{m-1} \beta_i u(k-i) \tag{2.2}$$

Model III:

$$y_p(k+1) = f[y_p(k), y_p(k-1), \ldots, y_p(k-n+1)] + g[u(k), u(k-1), \ldots, u(k-m+1)] \tag{2.3}$$

Model IV:

$$y_p(k+1) = f[y_p(k), y_p(k-1), \ldots, y_p(k-n+1), u(k), u(k-1), \ldots, u(k-m+1)] \tag{2.4}$$

where $[u(k), y_p(k)]$ represents the input-output of the model and f and g represent feedforward neural networks. The last two models facilitate neural network adaptive control [34]:

Narma-L1 Model:

$$\begin{aligned} y_p(k+d) &= f[y_p(k), y_p(k-1), \ldots, y_p(k-n+1)] \\ &\quad + \sum_{i=0}^{m-1} g_i[y_p(k), y_p(k-1), \ldots, y_p(k-n+1)] \cdot u(k-i) \end{aligned} \tag{2.5}$$

Narma-L2 Model:

$$\begin{aligned} y_p(k+d) &= f[y_p(k), y_p(k-1), \ldots, y_p(k-n+1), u(k-1), \ldots, u(k-n+1)] \\ &\quad + g_0[y_p(k), y_p(k-1), \ldots, y_p(k-n+1), u(k-1), \ldots, u(k-n+1)] \cdot u(k) \end{aligned} \tag{2.6}$$

Each f and g represent neural networks. Models like the NARMA-L2 allow the configuration of neurocontrollers. The plant is identified using Eq. (2.6) and solving for $u(k)$ we obtain:

$$u(k) = \frac{y_p(k+d) - f[y_p(k), y_p(k-1), \ldots, y_p(k-n+1), u(k-1), \ldots, u(k-n+1)]}{g_0[y_p(k), y_p(k-1), \ldots, y_p(k-n+1), u(k-1), \ldots, u(k-n+1)]} \tag{2.7}$$

Frasconi et al. [17] proposed an architecture where a feedforward neural network has a simple local feedback loop around each hidden layer. A more complex model called the Fully Connected Hidden Layer Recurrent Neural Network was proposed by Elman [14] and Mills et al. [31] where we have single delays between all layers.

The Finite Impulse Response (FIR) Neural Network [50, 3] replaces each network synaptic weight by an FIR filter:

$$y(t) = \sum_{i=1}^{n_b} w_i x(t-i) \tag{2.8}$$

where *x(t)* is the input to the filter, *y(t)* is the output of the filter and n_b is the filter order.

The Gamma Network [13, 39, 33, 24, 40] replaced the inputs to the hidden layers by gamma filters:

$$\begin{aligned} y_{i,0}(t) &= u(t) \\ y_{i,k}(t) &= (1-\mu_i)y_{i,k}(t-1) + \mu_i y_{i,k-1}(t-1) \end{aligned} \tag{2.9}$$

where $i = 1, \ldots, N$, $k = 1, \ldots, K$ and $t = 1, \ldots, t_f$. *N* is the dimension of the input signal *u(t)*, *K* is the order of the memory structure and $\mu_i's$ are the parameters of the memory banks. Figure 2.2 shows a Gamma Network where only one memory bank is shown in detail. Each memory consists of a tapped delay line with a local recurrent connection.

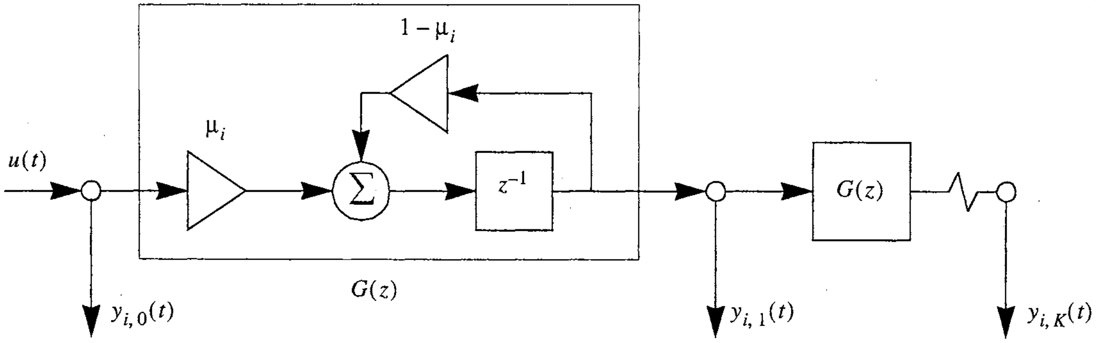

Figure 2.2: *The discrete Gamma Network.*

Another representation is the Infinite Impulse Response (IIR) Network [3, 37] where the weights are replaced by Infinite Impulse Response (IIR) filters. The direct form of an IIR filter is:

$$y(t) = \left(\frac{\sum_{i=1}^{M} b_i q^{-i}}{1 - \sum_{i=1}^{N} a_i q^{-i}} \right) u(t) \qquad \textbf{(2.10)}$$

or

$$y(t) = a_1 y(t-1) + a_2 y(t-2) + \ldots + a_N y(t-N) + b_1 u(t-1) + b_2 u(t-2) + \ldots + b_M u(t-M) \qquad \textbf{(2.11)}$$

where a_i, $i = 1, \ldots, N$ and b_i, $i = 1, \ldots, M$ are the weight parameters for the IIR filter for the layer and M and N are the filter orders.

Now that we have reviewed some typical dynamic networks, we next want to introduce a very general class of dynamic network. The *Layered Digital Dynamic Network* (LDDN) is a general neural network architecture able to represent all the previous specific architectures presented by different researchers. Each of the previous dynamic architectures has its own training algorithm; we will present general algorithms able to train arbitrary LDDNs. The LDDN is a generalization of the *Layered Feedforward Network* (LFFN), which has been modified to include feedback connections and delays. We begin here with a description of the LFFN, and then show how it can be generalized to obtain the LDDN.

2.3. Layered Feedforward Network.

Figure 2.3 is an example of a layered feedforward network (two layers in this case). (See [12] for a full description of the notation used here.) The input vector to the network is represented by $\mathbf{p}^1$, which has R^1 elements. The superscript represents the input number, since it is possible to have more than one input vector. The input is connected to Layer 1 through the input weight $\mathbf{IW}^{1,1}$, where the first superscript represents the layer number and the second superscript represents the input number. The bias for the first layer is represented by $\mathbf{b}^1$. The net input to Layer 1 is denoted by $\mathbf{n}^1$, and is computed as

$$\mathbf{n}^1 = \mathbf{IW}^{1,1}\mathbf{p}^1 + \mathbf{b}^1 \tag{2.12}$$

The output of Layer 1, $\mathbf{a}^1$, is computed by passing the net input through a transfer function, according to $\mathbf{a}^1 = \mathbf{f}^1(\mathbf{n}^1)$. The output has S^1 elements. The output of the first layer is input to the second layer through the layer weight $\mathbf{LW}^{2,1}$, where the first superscript represents the destination layer number and the second superscript represents the source layer number. The overall output of the network is labeled $\mathbf{y}$. This is typically chosen to be the output of the last layer in the network, as it is in Figure 2.3, although it could be the output of any layer in the network. Similarly, we could connect any input to any layer.

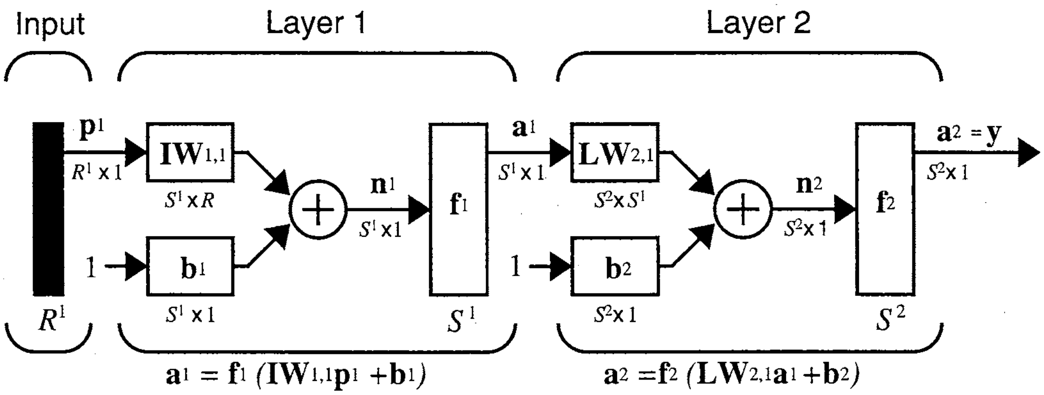

Figure 2.3: *Example of a Layered Feedforward Network*

Each layer in the LFFN is made up of:

1) a set of weight matrices that come into that layer (which may connect from other layers or from external inputs),

2) a bias vector,

3) a summing junction and

4) a transfer function.

In the example given in Figure 2.3, there is only one weight matrix associated with each layer, but it is possible to have weight matrices that are connected from several different input vectors and layer outputs. This will become clear when we introduce the LDDN network. Also, the example in Figure 2.3 has only two layers, our general LFFN can have an arbitrary number of layers. The layers do not have to be connected in sequence from Layer 1 to Layer M. For example, Layer 1 could be connected to both Layer 3 and Layer 4, by weights $\mathbf{LW}^{3,1}$ and $\mathbf{LW}^{4,1}$, respectively. Although the layers do not have to be connected in a linear sequence by layer number, it must be possible to compute the output of the network by a simple sequence of calculations. Due to the nature of the LFFN there cannot be any feedback loops in the network or feedforward delays. The order in which the individual layer outputs must be computed in order to obtain the correct network output is called the *simulation order*.

2.4. Layered Digital Dynamic Network.

We now introduce a class of dynamic networks that are based on the LFFN. The LFFN is a static network, in the sense that the network output can be computed directly from the network input, without the knowledge of initial network states. A Layered Digital Dynamic Network (LDDN) can contain feedback loops and time delays. The network response is a function of network inputs, as well as initial network states.

The components of the LDDN are the same as those of the LFFN, with the addition of the tapped delay line (TDL), which is shown in Figure 2.4. The output of the TDL is a vector containing current and/or previous values of the TDL input. In Figure 2.4 we show two abbreviated representations for the TDL. In the case on the left, the undelayed value of the input variable is included in the output vector. In the case on the right, only delayed values of the input are included in the output.

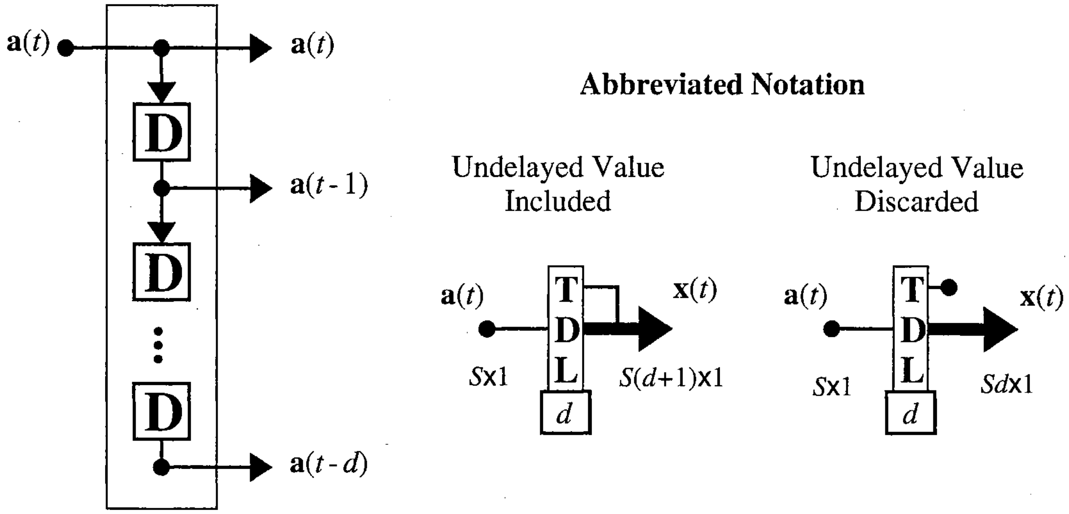

Figure 2.4: *Tapped Delay Line*

Figure 2.5 is an example of an LDDN. Like the LFFN, the LDDN is made up of layers. In addition to the weight matrices, bias, summing junction and transfer function, which make up the layers of the LFFN, the layers of the LDDN also include any tapped delay lines that appear at the input of a weight matrix, where any weight matrix in an LDDN can be proceeded by a tapped delay line. For example, Layer 1 of Figure 2.5 contains the weight $\mathbf{LW}^{1,2}$ and the TDL at its input. The output of the TDL in Figure 2.5 is labeled $\mathbf{\hat{a}}^{1,2}(t)$. This indicates that it is a composite vector made up of delayed values of the output of Layer 2 and is an input to Layer 1. These TDL outputs are important variables in our training algorithm for the LDDN. Note that all of the layer outputs and net inputs in the LDDN are explicit functions of time.

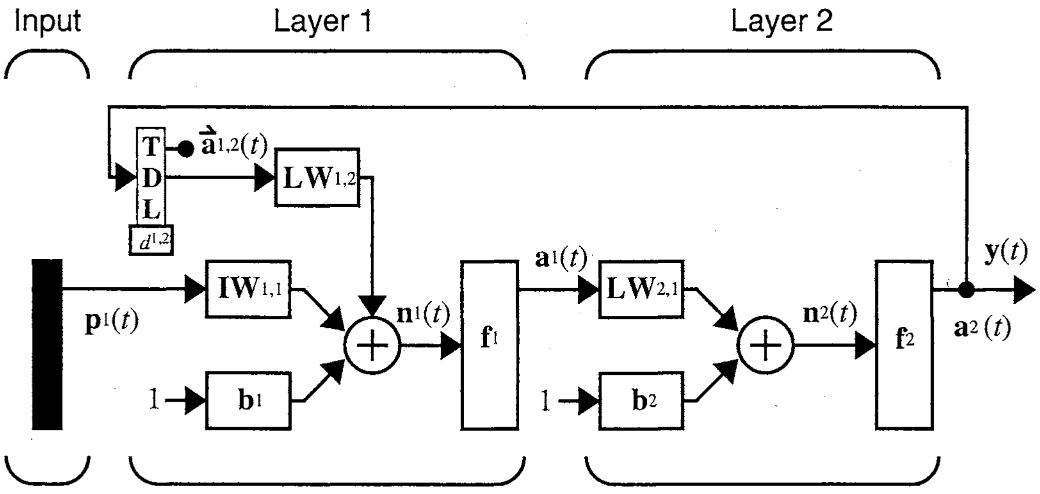

Figure 2.5: *Layered Digital Dynamic Network Example*

In the LDDN, feedback as well as feedforward delays are added to an LFFN. Therefore, unlike the LFFN, the output of the network is a function not only of the weights, biases, and network input, but also of the outputs of some of the network layers at previous points in time. For this reason, it is not a simple matter to calculate the gradient of the network output with respect to the weights and biases (which is needed to train the network). This is because the weights and biases have two different effects on the network output. The first is the direct effect, which can be calculated using the standard backpropagation algorithm [20]. The second is an indirect effect, since some of the inputs to the network, such as $\vec{\mathbf{a}}^{1,2}(t)$, are also functions of the weights and biases. In the next section we briefly describe the gradient calculations for the LFFN, and show how they must be modified for the LDDN. The main development of the next three chapters is a general gradient calculation for arbitrary LDDN's.

The LDDN architecture is very general and can be used to represent all of the dynamic networks that we reviewed at the beginning of this chapter. Appendix A demonstrates LDDN representations for several network types.

In the next three chapters we will develop algorithms for computing the gradient of LDDN errors with respect to the weights of the network. These will be general algorithms, applicable to any LDDN network. The gradients computed by these algorithms will then be used by optimization procedures to train the networks.

Chapter 3

DYNAMIC LEARNING

In this chapter we will introduce the basic concepts required for training dynamic networks. We begin with a review of gradient-based optimization algorithms that can be used to train both static and dynamic networks. We then describe how gradients can be computed for static networks. Finally, we use a simple dynamic network structure, with one feedback loop and one delay, to demonstrate gradient calculations for dynamic networks. (These calculations will be generalized for the LDDN in Chapter 4 and Chapter 5.)

3.1. Performance Optimization.

Once a neural network architecture is defined, the objective is to train the network. The training generally modifies the weights and biases to obtain a network that produces a specific behavior. We have three types of training algorithms: unsupervised learning, reinforcement or graded learning and supervised learning. *Unsupervised learning* is based only on the inputs of the neural network. *Reinforcement learning* is based on a grade or score for neural network performance. *Supervised learning* uses a set of examples of network inputs $\mathbf{p}_q$ and corresponding targets $\mathbf{t}_q$ [20]:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\} . \qquad \textbf{(3.1)}$$

This research will develop *supervised learning* algorithms for the LDDN. Due to the time dependencies of the LDDN, we will refer to the input-target sets as sequences. For static feedforward neural networks we can present data to the networks in batch form, where each input-target pair will be independent from other pairs. For dynamic systems, as we present the input sequence we must maintain the relative time position of each data point.

Supervised learning is based on the optimization of the performance of the neural network. A common *performance index* is the *sum squared error*:

$$F_{sse}(\mathbf{x}) = \sum_{q=1}^{Q} (\mathbf{t}_q - \mathbf{y}_q)^T (\mathbf{t}_q - \mathbf{y}_q) = \sum_{q=1}^{Q} \mathbf{e}_q^T \mathbf{e}_q \tag{3.2}$$

where $\mathbf{x}$ is the vector of network weights and biases, $\mathbf{t}_q$ is the target vector, $\mathbf{y}_q$ is the output vector when the qth input, $\mathbf{p}_q$, is presented and $\mathbf{e}_q = \mathbf{t}_q - \mathbf{y}_q$ is the error.

A second performance index is *sum squared error with regularization*:

$$F_{ssereg}(\mathbf{x}) = \alpha F_{sse}(\mathbf{x}) + (1-\alpha)[\mathbf{x}^T \cdot \mathbf{x}] \tag{3.3}$$

which combines sum squared error and sum squared weights and biases. The ratio α is a value between 0 and 1.

The optimization of any of the performance indexes begins with an initial set of weights $\mathbf{x}_0$ and the iterative update of the weights for each training epoch k according to:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mu_k \mathbf{d}_k \tag{3.4}$$

or

$$\Delta\mathbf{x}_k = \mathbf{x}_{k+1} - \mathbf{x}_k = \mu_k \mathbf{d}_k \tag{3.5}$$

where the vector $\mathbf{d}_k$ represents a search direction, and the positive scalar μ_k is the learning rate. The optimization algorithms will determine the search direction $\mathbf{d}_k$ as well as the learning rate μ_k.

3.1.1. Steepest Descent.

Consider the first-order Taylor series expansion of any performance index $F(\mathbf{x})$ around the guess $\mathbf{x}_k$:

$$F(\mathbf{x}_{k+1}) = F(\mathbf{x}_k + \Delta\mathbf{x}_k) = F(\mathbf{x}_k) + \mathbf{g}_k^T \Delta\mathbf{x}_k \tag{3.6}$$

where $\mathbf{g}_k$ is the gradient evaluated at the guess $\mathbf{x}_k$:

$$\mathbf{g}_k = \nabla F(\mathbf{x})|_{\mathbf{x} = \mathbf{x}_k} \tag{3.7}$$

For minimization, $F(\mathbf{x}_{k+1})$ should be smaller than $F(\mathbf{x}_k)$, so the second term on the right-hand side of Eq.

(3.6) must be negative:

$$\mathbf{g}_k^T \Delta \mathbf{x}_k = \mu_k \mathbf{g}_k^T \mathbf{d}_k < 0 \tag{3.8}$$

We will select a positive learning rate μ_k, so $\mathbf{g}_k^T \mathbf{d}_k < 0$. The performance function will decrease fastest when $\mathbf{g}_k^T \mathbf{d}_k$ is most negative, which will occur when the search direction vector $\mathbf{d}_k$ is the negative of the gradient:

$$\mathbf{d}_k = -\mathbf{g}_k \tag{3.9}$$

We can redefine Eq. (3.4) for the *steepest descent* method as:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mu_k \mathbf{g}_k \tag{3.10}$$

We have two methods to determine the learning rate μ_k for steepest descent:

1.- Minimize $F(\mathbf{x})$ along the line $\mathbf{x}_k - \mu_k \mathbf{g}_k$ for each iteration.

2.- Use a fixed value (for example $\mu_k = 0.02$) or use variable but predetermined values (for example $\mu_k = 1/k$).

3.1.2. Newton's Method.

This method is based on the second-order Taylor series approximation [20]:

$$F(\mathbf{x}_{k+1}) = F(\mathbf{x}_k + \Delta \mathbf{x}_k) = F(\mathbf{x}_k) + \mathbf{g}_k^T \Delta \mathbf{x}_k + \frac{1}{2} \Delta \mathbf{x}_k{}^T \mathbf{A}_k \Delta \mathbf{x}_k \tag{3.11}$$

To locate the stationary point of the previous approximation we take its gradient with respect to $\Delta \mathbf{x}_k$:

$$\nabla F(\mathbf{x}_{k+1}) = \mathbf{g}_k + \mathbf{A}_k \Delta \mathbf{x}_k , \tag{3.12}$$

and set it equal to zero. Solving for $\Delta \mathbf{x}_k$:

$$\Delta \mathbf{x}_k = -\mathbf{A}_k^{-1} \mathbf{g}_k \tag{3.13}$$

From the previous relation, we obtain *Newton's Method*:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1} \mathbf{g}_k \tag{3.14}$$

The drawback of Newton's Method is the calculation and storage of the Hessian Matrix $\mathbf{A}_k$, as well as its inverse. Two alternatives are the quasi-Newton methods and the one-step-secant methods. These methods replace $\mathbf{A}_k^{-1}$ with a positive definite matrix $\mathbf{H}_k$, that is updated at each iteration without matrix inversion.

3.1.3. Conjugate Gradient.

The calculation and storage of the second derivatives may impractical for the performance optimization of neural networks, especially for a large number of weights and biases. An alternative possibility that emulates a quadratic trajectory is conjugate directions [20].

For the quadratic performance function:

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{b}^T\mathbf{x} + c\,, \tag{3.15}$$

a set of vectors $\{\mathbf{d}_k\}$ is mutually *conjugate* with respect to a positive definite Hessian matrix $\mathbf{A}$ if and only if

$$\mathbf{d}_k^T\mathbf{A}\mathbf{d}_j = 0 \qquad k \neq j \tag{3.16}$$

For quadratic functions we have:

$$\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b} \tag{3.17}$$

$$\nabla^2 F(\mathbf{x}) = \mathbf{A} \tag{3.18}$$

We can combine the previous equations to find the change in gradient for the iteration $k + 1$:

$$\Delta\mathbf{g}_k = \mathbf{g}_{k+1} - \mathbf{g}_k = (\mathbf{A}\mathbf{x}_{k+1} + \mathbf{b}) - (\mathbf{A}\mathbf{x}_k + \mathbf{b}) = \mathbf{A}\Delta\mathbf{x} \tag{3.19}$$

By combining Eq. (3.5) and the conjugate condition Eq. (3.16):

$$\mu_k\mathbf{d}_k^T\mathbf{A}\mathbf{d}_j = \Delta\mathbf{x}_k^T\mathbf{A}\mathbf{d}_j = \Delta\mathbf{g}_k^T\mathbf{d}_j = 0 \qquad k \neq j \tag{3.20}$$

From the previous relation, the Hessian matrix is no longer needed. The conjugate conditions could be obtained recursively by starting the search in the steepest descent direction:

$$\mathbf{d}_0 = -\mathbf{g}_0 \tag{3.21}$$

and updating the search direction by:

$$\mathbf{d}_k = -\mathbf{g}_k + \beta_k\mathbf{d}_{k-1} \tag{3.22}$$

The most common choices to select the scalar value β_k are [43]:

$$\beta_k = \frac{\Delta\mathbf{g}_{k-1}^T\mathbf{g}_k}{\Delta\mathbf{g}_{k-1}^T\mathbf{d}_{k-1}}\,, \tag{3.23}$$

due to Hestenes and Steifel,

$$\beta_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}, \tag{3.24}$$

due to Fletcher and Reeves, and

$$\beta_k = \frac{\Delta\mathbf{g}_{k-1}^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}, \tag{3.25}$$

due to Polak and Ribiére.

3.1.4. Levenberg-Marquardt Algorithm.

This algorithm is a variation of Newton's method based on the sum squared error (Eq. (3.2)) [21, 20]. From Eq. (3.14) Newton's method for optimizing a performance index $F(\mathbf{x})$ is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1}\mathbf{g}_k \tag{3.26}$$

where $\mathbf{A}_k \equiv \nabla^2 F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}_k}$ and $\mathbf{g}_k \equiv \nabla F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}_k}$

For a performance function like the sum squared error:

$$F(\mathbf{x}) = \sum_{i=1}^{N} v_i^2(\mathbf{x}) = \mathbf{v}^T(\mathbf{x})\mathbf{v}(\mathbf{x}), \tag{3.27}$$

the jth element of the gradient is

$$[\nabla F(\mathbf{x})]_j = \frac{\partial F(\mathbf{x})}{\partial x_j} = 2\sum_{i=1}^{N} v_i(\mathbf{x})\frac{\partial v_i(\mathbf{x})}{\partial x_j}. \tag{3.28}$$

We can rewrite the gradient in matrix form:

$$\nabla F(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{v}(\mathbf{x}), \tag{3.29}$$

where

$$\mathbf{J}^T(\mathbf{x}) = \begin{bmatrix} \frac{\partial v_1(\mathbf{x})}{\partial x_1} & \frac{\partial v_1(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial v_1(\mathbf{x})}{\partial x_n} \\ \frac{\partial v_2(\mathbf{x})}{\partial x_1} & \frac{\partial v_2(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial v_2(\mathbf{x})}{\partial x_n} \\ \cdots & \cdots & & \cdots \\ \frac{\partial v_N(\mathbf{x})}{\partial x_1} & \frac{\partial v_N(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial v_N(\mathbf{x})}{\partial x_n} \end{bmatrix} \tag{3.30}$$

is the *Jacobian matrix*. The *k, j* term of the Hessian matrix would be

$$[\nabla^2 F(\mathbf{x})]_{k,j} = \frac{\partial^2 F(\mathbf{x})}{\partial x_j \partial x_j} = 2\sum_{i=1}^{N}\left\{\frac{\partial v_i(\mathbf{x})}{\partial x_k}\frac{\partial v_i(\mathbf{x})}{\partial x_j} + v_i(\mathbf{x})\frac{\partial^2 v_i(\mathbf{x})}{\partial x_j \partial x_j}\right\}, \tag{3.31}$$

then we can represent the Hessian in matrix form:

$$\nabla^2 F(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}) + 2\mathbf{S}(\mathbf{x}), \tag{3.32}$$

where

$$\mathbf{S}(\mathbf{x}) = \sum_{i=1}^{N} v_i(\mathbf{x})\nabla^2 v_i(\mathbf{x}) \tag{3.33}$$

If we assume that $\mathbf{S}(\mathbf{x})$ is small, we can approximate the Hessian matrix as

$$\nabla^2 F(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}). \tag{3.34}$$

By substitution of Eq. (3.34) and Eq. (3.29) into Eq. (3.26) we obtain the Gauss-Newton method:

$$\begin{aligned}\mathbf{x}_{k+1} &= \mathbf{x}_k - [2\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k)]^{-1}2\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k)\\ &= \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k)]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k)\end{aligned} \tag{3.35}$$

The advantage of the *Gauss-Newton* method over Newton's method (Eq. (3.14)) is that we only require the calculation of first derivatives. The drawback of the Gauss-Newton method is that the matrix $\mathbf{H} = \mathbf{J}^T\mathbf{J}$ may not have inverse. We can overcome this problem by using the approximation:

$$\mathbf{G} = \mathbf{H} + \mu\mathbf{I}, \tag{3.36}$$

The eigenvalues and eigenvectors of **H** are $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ and $\{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n\}$, so we have:

$$\mathbf{G}\mathbf{z}_i = [\mathbf{H} + \mu\mathbf{I}]\mathbf{z}_i = \mathbf{H}\mathbf{z}_i + \mu\mathbf{z}_i = \lambda_i\mathbf{z}_i + \mu\mathbf{z}_i = (\lambda_i + \mu)\mathbf{z}_i, \tag{3.37}$$

meaning that **G** and **H** have the same eigenvectors, and the eigenvalues of **G** are $(\lambda_i + \mu)$. We can increase μ until $(\lambda_i + \mu) > 0$ for all *i*, allowing us to invert the matrix **G**.

The **G** approximation leads to the *Levenberg-Marquardt* algorithm [43, 21, 20]:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \mu_k\mathbf{I}]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k) \tag{3.38}$$

or

$$\Delta \mathbf{x}_k = -[\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \mu_k \mathbf{I}]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k) . \tag{3.39}$$

This algorithm approximates the steepest descent algorithm for large μ_k and approximates the Gauss-Newton method for small μ_k.

3.2. Gradient Calculations for Static Networks.

In order to use the optimization algorithms described in the previous section, we need to be able to compute the gradients of the performance index with respect to the weights and biases. In this section we will demonstrate the gradient calculations for static networks.

Consider again the multilayer network of Figure 2.3 on page 8. The basic simulation equation of such a network is

$$\mathbf{a}^k = \mathbf{f}^k\left(\sum_i \mathbf{IW}^{k,i}\mathbf{p}^i + \sum_j \mathbf{LW}^{k,j}\mathbf{a}^j + \mathbf{b}^k\right), \tag{3.40}$$

where k is incremented through the *simulation order*. This is the order that will make the necessary inputs at each layer available, as will be discussed later. The summation over i represents the inputs connected to layer k and the summation over j represents the layers connected to layer k.

We want to compute the gradient of the sum squared error performance index:

$$F(\mathbf{x}) = \sum_{q=1}^{Q} (\mathbf{t}_q - \mathbf{y}_q)^T(\mathbf{t}_q - \mathbf{y}_q) = \sum_{q=1}^{Q} \mathbf{e}_q{}^T\mathbf{e}_q \tag{3.41}$$

The gradient of the sum squared error is the sum of the gradients of each individual squared error [20]:

$$\hat{F} = \mathbf{e}_q{}^T\mathbf{e}_q, \tag{3.42}$$

so we will consider these terms individually. We want to compute the terms

$$\frac{\partial \hat{F}}{\partial w_{i,j}^{k,l}} \text{ and } \frac{\partial \hat{F}}{\partial b_i^k} \tag{3.43}$$

Define

$$s_i^k \equiv \frac{\partial \hat{F}}{\partial n_i^k} \tag{3.44}$$

as the *sensitivity* of the performance index to changes in the net input of unit *i* in layer *k*. Using the chain rule, we can show that

$$\frac{\partial \hat{F}}{\partial iw_{i,j}^{m,l}} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial iw_{i,j}^{m,l}} = s_i^m p_j^l, \quad \frac{\partial \hat{F}}{\partial lw_{i,j}^{m,l}} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial lw_{i,j}^{m,l}} = s_i^m a_j^l, \quad \frac{\partial \hat{F}}{\partial b_i^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m} = s_i^m \tag{3.45}$$

It can also be shown that the sensitivities satisfy the following recurrence relation, in which *m* is incremented through the *backpropagation order*, which is the reverse of the simulation order:

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)\sum_i (\mathbf{LW}^{i,m})^T \mathbf{s}^i \tag{3.46}$$

where

$$\dot{\mathbf{F}}^m(\mathbf{n}^m) = \begin{bmatrix} \dot{f}^m(n_1^m) & 0 & \dots & 0 \\ 0 & \dot{f}^m(n_2^m) & \dots & 0 \\ \vdots & \vdots & \dots & \vdots \\ 0 & 0 & \dots & \dot{f}^m(n_{S^m}^m) \end{bmatrix} \tag{3.47}$$

and

$$\dot{f}^m(n_j^m) = \frac{\partial f^m(n_j^m)}{\partial n_j^m} \tag{3.48}$$

This recurrence relation is initialized at the output layer:

$$\mathbf{s}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t}_q - \mathbf{y}_q)\,. \tag{3.49}$$

The overall algorithm now proceeds as follows: first, propagate the input forward using Eq. (3.40); next, propagate the sensitivities back using Eq. (3.49) and Eq. (3.46); and finally, compute the gradient using Eq. (3.45).

3.3. Dynamic Training Principles.

Now consider an LDDN, such as the one shown in Figure 2.5 on page 11. Suppose that we want to compute the same gradient, Eq. (3.43), that is computed by the standard backpropagation algorithm. The problem in this case is that when we try to find the equivalent of Eq. (3.45) we note that the weights and biases have two different effects on the network output. The first is the direct effect, which is accounted for by Eq.

(3.45). The second is an indirect effect, since some of the inputs to the network, such as $\mathbf{a}^{1,2}(t)$, are also functions of the weights and biases. To account for this indirect effect we must use *dynamic backpropagation.*

To illustrate dynamic backpropagation [55, 56, 19], consider Figure 3.1, which is a simple dynamic network. It consists of an LFFN with a single feedback loop added from the output of the network, which is connected to the input of the network through a single delay. In this figure the vector $\mathbf{x}$ represents all of the network parameters (weights and biases) and the vector $\mathbf{a}(t)$ represents the output of the LFFN at time step t.

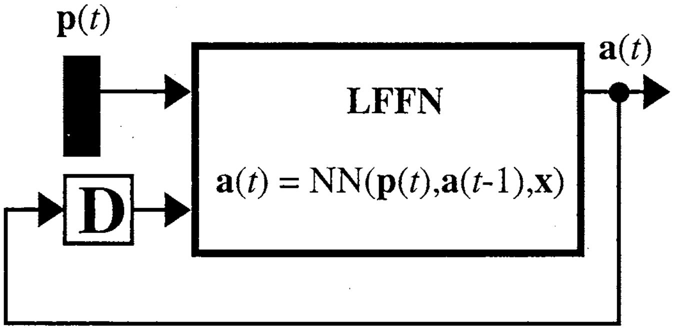

Figure 3.1: *Simple dynamic network*

Now suppose that we want to minimize

$$F(\mathbf{x}) = \sum_{t=1}^{Q} (\mathbf{t}(t) - \mathbf{a}(t))^T(\mathbf{t}(t) - \mathbf{a}(t)) \tag{3.50}$$

In order to use gradient descent, we need to find the gradient of F with respect to the network parameters. There are two different approaches to this problem. They both use the chain rule, but are implemented in different ways:

$$\frac{\partial F}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \left[\frac{\partial \mathbf{a}(t)}{\partial \mathbf{x}}\right]^T \times \frac{\partial^e F}{\partial \mathbf{a}(t)} \tag{3.51}$$

or

$$\frac{\partial F}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \left[\frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}}\right]^T \times \frac{\partial F}{\partial \mathbf{a}(t)} \tag{3.52}$$

where the superscript e indicates an explicit derivative, not accounting for indirect effects through time. The explicit derivatives can be obtained with the standard backpropagation algorithm, as in Eq. (3.46) [20]. To find the complete derivatives that are required in Eq. (3.51) and Eq. (3.52), we need the additional equations:

$$\frac{\partial \mathbf{a}(t)}{\partial \mathbf{x}} = \frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}} + \frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{a}(t-1)} \times \frac{\partial \mathbf{a}(t-1)}{\partial \mathbf{x}} \tag{3.53}$$

and

$$\frac{\partial F}{\partial \mathbf{a}(t)} = \frac{\partial^e F}{\partial \mathbf{a}(t)} + \frac{\partial^e \mathbf{a}(t+1)}{\partial \mathbf{a}(t)} \times \frac{\partial F}{\partial \mathbf{a}(t+1)} \tag{3.54}$$

Eq. (3.51) and Eq. (3.53) make up the forward perturbation (FP) algorithm [54]. Note that the key term is

$$\frac{\partial \mathbf{a}(t)}{\partial \mathbf{x}} \tag{3.55}$$

which must be propagated forward through time.

Eq. (3.52) and Eq. (3.54) make up the backpropagation-through-time (BTT) algorithm [51]. Here the key term is

$$\frac{\partial F}{\partial \mathbf{a}(t)} \tag{3.56}$$

which must be propagated backward through time.

In general, the FP algorithm requires somewhat more computation than the BTT algorithm. However, the BTT algorithm cannot be implemented in real-time, since the outputs must be computed for all time steps, and then the derivatives must be backpropagated back to the initial time point. The FP algorithm is well-suited for real-time implementation, since the derivatives can be calculated at each time step.

3.4. Additional methods for dynamic gradient calculation.

Additional dynamic gradient calculation methods had been proposed in the literature. Generally those methods are derived from the BP and the BTT algorithms. The Green's Function method [45, 2] is a simplification of the FP algorithm. The idea is to obtain the gradient $\partial F/\partial\mathbf{x}$ for a sequence of Q terms based on the previously calculated gradient for *Q-1* terms. If we subtract both gradient values based on Eq. (3.51):

$$\begin{aligned}
\left.\frac{\partial F}{\partial\mathbf{x}}\right|_{Q} - \left.\frac{\partial F}{\partial\mathbf{x}}\right|_{Q-1} &= \left[\frac{\partial\mathbf{a}(Q)}{\partial\mathbf{x}}\right]^{T}\times\frac{\partial^{e}F}{\partial\mathbf{a}(Q)}\\
&= \left[\frac{\partial^{e}\mathbf{a}(Q)}{\partial\mathbf{x}}+\frac{\partial^{e}\mathbf{a}(Q)}{\partial\mathbf{a}(Q-1)}\times\frac{\partial\mathbf{a}(Q-1)}{\partial\mathbf{x}}\right]^{T}\times\frac{\partial^{e}F}{\partial\mathbf{a}(Q)}\\
&= \left[\frac{\partial^{e}\mathbf{a}(Q)}{\partial\mathbf{x}}+\frac{\partial^{e}\mathbf{a}(Q)}{\partial\mathbf{a}(Q-1)}\times\left(\frac{\partial^{e}\mathbf{a}(Q-1)}{\partial\mathbf{x}}+\frac{\partial^{e}\mathbf{a}(Q-1)}{\partial\mathbf{a}(Q-2)}\times\left(\frac{\partial^{e}\mathbf{a}(Q-2)}{\partial\mathbf{x}}\right.\right.\right.\\
&\quad\left.\left.\left.+\frac{\partial^{e}\mathbf{a}(Q-2)}{\partial\mathbf{a}(Q-3)}\times\left(\ldots\times\left(\frac{\partial^{e}\mathbf{a}(2)}{\partial\mathbf{x}}+\frac{\partial^{e}\mathbf{a}(2)}{\partial\mathbf{a}(1)}\times\frac{\partial^{e}\mathbf{a}(1)}{\partial\mathbf{x}}\right)\ldots\right)\right)\right)\right]^{T}\times\frac{\partial^{e}F}{\partial\mathbf{a}(Q)}\\
&= \left[\frac{\partial^{e}\mathbf{a}(Q)}{\partial\mathbf{x}}+\frac{\partial^{e}\mathbf{a}(Q)}{\partial\mathbf{a}(Q-1)}\times\frac{\partial^{e}\mathbf{a}(Q-1)}{\partial\mathbf{x}}+\frac{\partial^{e}\mathbf{a}(Q)}{\partial\mathbf{a}(Q-1)}\times\frac{\partial^{e}\mathbf{a}(Q-1)}{\partial\mathbf{a}(Q-2)}\times\frac{\partial^{e}\mathbf{a}(Q-2)}{\partial\mathbf{x}}\right.\\
&\quad\left.+\ldots+\frac{\partial^{e}\mathbf{a}(Q)}{\partial\mathbf{a}(Q-1)}\times\frac{\partial^{e}\mathbf{a}(Q-1)}{\partial\mathbf{a}(Q-2)}\times\ldots\times\frac{\partial^{e}\mathbf{a}(2)}{\partial\mathbf{a}(1)}\times\frac{\partial^{e}\mathbf{a}(1)}{\partial\mathbf{x}}\right]^{T}\times\frac{\partial^{e}F}{\partial\mathbf{a}(Q)}
\end{aligned}\qquad\textbf{(3.57)}$$

We can multiply inside the transpose term by:

$$I = \frac{\partial^{e}\mathbf{a}(Q)}{\partial\mathbf{a}(Q-1)}\times\left(\frac{\partial^{e}\mathbf{a}(Q)}{\partial\mathbf{a}(Q-1)}\right)^{-1}$$

we will obtain:

$$\begin{aligned}
\left.\frac{\partial F}{\partial\mathbf{x}}\right|_{Q} - \left.\frac{\partial F}{\partial\mathbf{x}}\right|_{Q-1} &= \left[\frac{\partial^{e}\mathbf{a}(Q)}{\partial\mathbf{a}(Q-1)}\times\left(\left(\frac{\partial^{e}\mathbf{a}(Q)}{\partial\mathbf{a}(Q-1)}\right)^{-1}\times\frac{\partial^{e}\mathbf{a}(Q)}{\partial\mathbf{x}}+\frac{\partial^{e}\mathbf{a}(Q-1)}{\partial\mathbf{x}}+\frac{\partial^{e}\mathbf{a}(Q-1)}{\partial\mathbf{a}(Q-2)}\times\frac{\partial^{e}\mathbf{a}(Q-2)}{\partial\mathbf{x}}\right.\right.\\
&\quad\left.\left.+\ldots+\frac{\partial^{e}\mathbf{a}(Q-1)}{\partial\mathbf{a}(Q-2)}\times\ldots\times\frac{\partial^{e}\mathbf{a}(2)}{\partial\mathbf{a}(1)}\times\frac{\partial^{e}\mathbf{a}(1)}{\partial\mathbf{x}}\right)\right]^{T}\times\frac{\partial^{e}F}{\partial\mathbf{a}(Q)}
\end{aligned}\qquad\textbf{(3.58)}$$

If we continue until $I = \frac{\partial^{e}\mathbf{a}(2)}{\partial\mathbf{a}(1)}\times\left(\frac{\partial^{e}\mathbf{a}(2)}{\partial\mathbf{a}(1)}\right)^{-1}$:

$$\begin{aligned}
\left.\frac{\partial F}{\partial\mathbf{x}}\right|_{Q} - \left.\frac{\partial F}{\partial\mathbf{x}}\right|_{Q-1} &= \left[\left(\frac{\partial^{e}\mathbf{a}(Q)}{\partial\mathbf{a}(Q-1)}\times\frac{\partial^{e}\mathbf{a}(Q-1)}{\partial\mathbf{a}(Q-2)}\times\ldots\times\frac{\partial^{e}\mathbf{a}(3)}{\partial\mathbf{a}(2)}\times\frac{\partial^{e}\mathbf{a}(2)}{\partial\mathbf{a}(1)}\right)\right.\\
&\quad\times\left(\left(\frac{\partial^{e}\mathbf{a}(2)}{\partial\mathbf{a}(1)}\right)^{-1}\times\left(\frac{\partial^{e}\mathbf{a}(3)}{\partial\mathbf{a}(2)}\right)^{-1}\times\ldots\times\left(\frac{\partial^{e}\mathbf{a}(Q-1)}{\partial\mathbf{a}(Q-2)}\right)^{-1}\times\left(\frac{\partial^{e}\mathbf{a}(Q)}{\partial\mathbf{a}(Q-1)}\right)^{-1}\times\frac{\partial^{e}\mathbf{a}(Q)}{\partial\mathbf{x}}\right.\\
&\quad+\left(\frac{\partial^{e}\mathbf{a}(2)}{\partial\mathbf{a}(1)}\right)^{-1}\times\left(\frac{\partial^{e}\mathbf{a}(3)}{\partial\mathbf{a}(2)}\right)^{-1}\times\ldots\times\left(\frac{\partial^{e}\mathbf{a}(Q-1)}{\partial\mathbf{a}(Q-2)}\right)^{-1}\times\frac{\partial^{e}\mathbf{a}(Q-1)}{\partial\mathbf{x}}\\
&\quad\left.\left.+\ldots+\left(\frac{\partial^{e}\mathbf{a}(2)}{\partial\mathbf{a}(1)}\right)^{-1}\times\frac{\partial^{e}\mathbf{a}(2)}{\partial\mathbf{x}}+\frac{\partial^{e}\mathbf{a}(1)}{\partial\mathbf{x}}\right)\right]^{T}\times\frac{\partial^{e}F}{\partial\mathbf{a}(Q)}
\end{aligned}\qquad\textbf{(3.59)}$$

We could define:

$$\begin{aligned} U(Q) &= \frac{\partial^e \mathbf{a}(Q)}{\partial \mathbf{a}(Q-1)} \times \frac{\partial^e \mathbf{a}(Q-1)}{\partial \mathbf{a}(Q-2)} \times \ldots \times \frac{\partial^e \mathbf{a}(3)}{\partial \mathbf{a}(2)} \times \frac{\partial^e \mathbf{a}(2)}{\partial \mathbf{a}(1)} \\ &= \frac{\partial^e \mathbf{a}(Q)}{\partial \mathbf{a}(Q-1)} \times U(Q-1) \end{aligned} \tag{3.60}$$

$$\begin{aligned} S(Q) &= \left(\frac{\partial^e \mathbf{a}(2)}{\partial \mathbf{a}(1)}\right)^{-1} \times \left(\frac{\partial^e \mathbf{a}(3)}{\partial \mathbf{a}(2)}\right)^{-1} \times \ldots \times \left(\frac{\partial^e \mathbf{a}(Q-1)}{\partial \mathbf{a}(Q-2)}\right)^{-1} \times \left(\frac{\partial^e \mathbf{a}(Q)}{\partial \mathbf{a}(Q-1)}\right)^{-1} \times \frac{\partial^e \mathbf{a}(Q)}{\partial \mathbf{x}} \\ &\quad + \left(\frac{\partial^e \mathbf{a}(2)}{\partial \mathbf{a}(1)}\right)^{-1} \times \left(\frac{\partial^e \mathbf{a}(3)}{\partial \mathbf{a}(2)}\right)^{-1} \times \ldots \times \left(\frac{\partial^e \mathbf{a}(Q-1)}{\partial \mathbf{a}(Q-2)}\right)^{-1} \times \frac{\partial^e \mathbf{a}(Q-1)}{\partial \mathbf{x}} \\ &\quad + \ldots + \left(\frac{\partial^e \mathbf{a}(2)}{\partial \mathbf{a}(1)}\right)^{-1} \times \frac{\partial^e \mathbf{a}(2)}{\partial \mathbf{x}} + \frac{\partial^e \mathbf{a}(1)}{\partial \mathbf{x}} \\ &= (U(Q))^{-1} \times \frac{\partial^e \mathbf{a}(Q)}{\partial \mathbf{x}} + S(Q-1) \end{aligned} \tag{3.61}$$

Finally we can update the gradient using:

$$\left.\frac{\partial F}{\partial \mathbf{x}}\right|_{Q} = \left.\frac{\partial F}{\partial \mathbf{x}}\right|_{Q-1} + [U(Q) \times S(Q)]^T \times \frac{\partial^e F}{\partial \mathbf{a}(Q)} \tag{3.62}$$

The Fast Forward Propagation Method [46, 47, 2] calculates the gradient recursively, allowing an on-line version of the BTT method. Let us assume that we can rewrite Eq. (3.54) as:

$$\frac{\partial F}{\partial \mathbf{a}(k)} = \frac{\partial^e F}{\partial \mathbf{a}(1)} \times A(k) + b(k) \ . \tag{3.63}$$

If we solve recursively:

$$\begin{aligned} \frac{\partial F}{\partial \mathbf{a}(Q)} &= \frac{\partial^e F}{\partial \mathbf{a}(Q)} + \cancel{\frac{\partial^e \mathbf{a}(Q+1)}{\partial \mathbf{a}(Q)} \times \frac{\partial F}{\partial \mathbf{a}(Q+1)}} \\ \frac{\partial F}{\partial \mathbf{a}(Q-1)} &= \frac{\partial^e F}{\partial \mathbf{a}(Q-1)} + \frac{\partial^e \mathbf{a}(Q)}{\partial \mathbf{a}(Q-1)} \times \frac{\partial F}{\partial \mathbf{a}(Q)} \\ &\ldots \\ \frac{\partial F}{\partial \mathbf{a}(2)} &= \frac{\partial^e F}{\partial \mathbf{a}(2)} + \frac{\partial^e \mathbf{a}(3)}{\partial \mathbf{a}(2)} \times \frac{\partial F}{\partial \mathbf{a}(3)} \\ \frac{\partial F}{\partial \mathbf{a}(1)} &= \frac{\partial^e F}{\partial \mathbf{a}(1)} + \frac{\partial^e \mathbf{a}(2)}{\partial \mathbf{a}(1)} \times \frac{\partial F}{\partial \mathbf{a}(2)} \end{aligned} \tag{3.64}$$

we have that:

$$\begin{aligned}
\left.\frac{\partial F}{\partial \mathbf{a}(1)}\right|_Q &= \frac{\partial^e F}{\partial \mathbf{a}(1)} + \frac{\partial^e \mathbf{a}(2)}{\partial \mathbf{a}(1)} \times \left(\frac{\partial^e F}{\partial \mathbf{a}(2)} + \frac{\partial^e \mathbf{a}(3)}{\partial \mathbf{a}(2)} \times \left(\ldots\left(\frac{\partial^e F}{\partial \mathbf{a}(Q-1)} + \frac{\partial^e \mathbf{a}(Q)}{\partial \mathbf{a}(Q-1)} \times \frac{\partial^e F}{\partial \mathbf{a}(Q)}\right)\right)\right) \\
&= \frac{\partial^e F}{\partial \mathbf{a}(1)} + \frac{\partial^e \mathbf{a}(2)}{\partial \mathbf{a}(1)} \times \frac{\partial^e F}{\partial \mathbf{a}(2)} + \ldots + \frac{\partial^e \mathbf{a}(2)}{\partial \mathbf{a}(1)} \times \frac{\partial^e \mathbf{a}(3)}{\partial \mathbf{a}(2)} \times \ldots \times \frac{\partial^e \mathbf{a}(Q)}{\partial \mathbf{a}(Q-1)} \times \frac{\partial^e F}{\partial \mathbf{a}(Q)} \quad . \\
&= \left.\frac{\partial F}{\partial \mathbf{a}(1)}\right|_{Q-1} + \frac{\partial^e \mathbf{a}(2)}{\partial \mathbf{a}(1)} \times \frac{\partial^e \mathbf{a}(3)}{\partial \mathbf{a}(2)} \times \ldots \times \frac{\partial^e \mathbf{a}(Q)}{\partial \mathbf{a}(Q-1)} \times \frac{\partial^e F}{\partial \mathbf{a}(Q)}
\end{aligned} \tag{3.65}$$

For a generic time $t=k$ we have:

$$\begin{aligned}
\left.\frac{\partial F}{\partial \mathbf{a}(k)}\right|_Q &= \left(\frac{\partial^e \mathbf{a}(k)}{\partial \mathbf{a}(k-1)}\right)^{-1} \times \left(-\frac{\partial^e F}{\partial \mathbf{a}(k-1)} + \left(\frac{\partial^e \mathbf{a}(k-1)}{\partial \mathbf{a}(k-2)}\right)^{-1} \times (\ldots\right. \\
&\qquad \left. \times\left(-\frac{\partial^e F}{\partial \mathbf{a}(3)} + \left(\frac{\partial^e \mathbf{a}(3)}{\partial \mathbf{a}(2)}\right)^{-1} \times \left(-\frac{\partial^e F}{\partial \mathbf{a}(2)} + \left(\frac{\partial^e \mathbf{a}(2)}{\partial \mathbf{a}(1)}\right)^{-1} \times \left(-\frac{\partial^e F}{\partial \mathbf{a}(1)} + \left.\frac{\partial F}{\partial \mathbf{a}(1)}\right|_Q\right)\right)\right)\ldots\right) \\
&= -\left(\frac{\partial^e \mathbf{a}(k)}{\partial \mathbf{a}(k-1)}\right)^{-1} \times \frac{\partial^e F}{\partial \mathbf{a}(k-1)} - \left(\frac{\partial^e \mathbf{a}(k)}{\partial \mathbf{a}(k-1)}\right)^{-1} \times \left(\frac{\partial^e \mathbf{a}(k-1)}{\partial \mathbf{a}(k-2)}\right)^{-1} \times \frac{\partial^e F}{\partial \mathbf{a}(k-2)} - \ldots \\
&\qquad -\left(\frac{\partial^e \mathbf{a}(k)}{\partial \mathbf{a}(k-1)}\right)^{-1} \times \left(\frac{\partial^e \mathbf{a}(k-1)}{\partial \mathbf{a}(k-2)}\right)^{-1} \times \ldots \times \left(\frac{\partial^e \mathbf{a}(2)}{\partial \mathbf{a}(1)}\right)^{-1} \times \frac{\partial^e F}{\partial \mathbf{a}(1)} \\
&\qquad +\left(\frac{\partial^e \mathbf{a}(k)}{\partial \mathbf{a}(k-1)}\right)^{-1} \times \left(\frac{\partial^e \mathbf{a}(k-1)}{\partial \mathbf{a}(k-2)}\right)^{-1} \times \ldots \times \left(\frac{\partial^e \mathbf{a}(2)}{\partial \mathbf{a}(1)}\right)^{-1} \times \left.\frac{\partial F}{\partial \mathbf{a}(1)}\right|_Q \\
&= b(k) + A(k) \times \left.\frac{\partial F}{\partial \mathbf{a}(1)}\right|_Q
\end{aligned} \tag{3.66}$$

where:

$$\begin{aligned}
b(k) &= -\left(\frac{\partial^e \mathbf{a}(k)}{\partial \mathbf{a}(k-1)}\right)^{-1} \times \frac{\partial^e F}{\partial \mathbf{a}(k-1)} - \left(\frac{\partial^e \mathbf{a}(k)}{\partial \mathbf{a}(k-1)}\right)^{-1} \times \left(\frac{\partial^e \mathbf{a}(k-1)}{\partial \mathbf{a}(k-2)}\right)^{-1} \times \frac{\partial^e F}{\partial \mathbf{a}(k-2)} - \ldots \\
&\qquad -\left(\frac{\partial^e \mathbf{a}(k)}{\partial \mathbf{a}(k-1)}\right)^{-1} \times \left(\frac{\partial^e \mathbf{a}(k-1)}{\partial \mathbf{a}(k-2)}\right)^{-1} \times \ldots \times \left(\frac{\partial^e \mathbf{a}(2)}{\partial \mathbf{a}(1)}\right)^{-1} \times \frac{\partial^e F}{\partial \mathbf{a}(1)} \\
&= \left(\frac{\partial^e \mathbf{a}(k)}{\partial \mathbf{a}(k-1)}\right)^{-1} \times \left(b(k-1) - \frac{\partial^e F}{\partial \mathbf{a}(k-1)}\right)
\end{aligned} \tag{3.67}$$

$$\begin{aligned}
A(k) &= \left(\frac{\partial^e \mathbf{a}(k)}{\partial \mathbf{a}(k-1)}\right)^{-1} \times \left(\frac{\partial^e \mathbf{a}(k-1)}{\partial \mathbf{a}(k-2)}\right)^{-1} \times \ldots \times \left(\frac{\partial^e \mathbf{a}(2)}{\partial \mathbf{a}(1)}\right)^{-1} \\
&= \left(\frac{\partial^e \mathbf{a}(k)}{\partial \mathbf{a}(k-1)}\right)^{-1} \times A(k-1)
\end{aligned} \tag{3.68}$$

allowing us to create an expression similar to Eq. (3.63). We can rewrite Eq. (3.52) using the final result of

Eq. (3.66) as:

$$\begin{aligned}\frac{\partial F}{\partial \mathbf{x}} &= \sum_{t=1}^{Q}\left[\frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}}\right]^T \times \left[b(t) + A(t) \times \left.\frac{\partial F}{\partial \mathbf{a}(1)}\right|_Q\right] \\ &= \sum_{t=1}^{Q}\left[\frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}}\right]^T \times b(t) + \sum_{t=1}^{Q}\left[\frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}}\right]^T \times A(t) \times \left.\frac{\partial F}{\partial \mathbf{a}(1)}\right|_Q\end{aligned} \quad . \tag{3.69}$$

The algorithm will work by solving recursively for $\left.\frac{\partial F}{\partial \mathbf{a}(1)}\right|_Q$ and $\left.\frac{\partial F}{\partial \mathbf{a}(k)}\right|_Q$, using the final results of Eq. (3.65) through Eq. (3.68) and by substituting into Eq. (3.69).

The Block Update method [44, 53, 2] updates the gradient every *N* steps. This algorithm combines characteristics of the FP and BTT and is well suited for very long sequences. Assume that we computed the gradient for a time *t=Q-N*. The idea is to wait until the time *t=Q* to update the gradient again. The gradient update will be:

$$G(Q) = \left.\frac{\partial F}{\partial \mathbf{x}}\right|_Q - \left.\frac{\partial F}{\partial \mathbf{x}}\right|_{Q-N} \tag{3.70}$$

where the gradient update *G(Q)* is calculated assuming that the term $\frac{\partial^e F}{\partial \mathbf{a}(t)}$ in Eq. (3.54) is zero for $t \le Q - N$. We can say that:

$$G(Q) = Z(1, Q-N) + Z(Q-N+1, Q) \tag{3.71}$$

where:

$$Z(q_1, q_2) = \sum_{t=q_1}^{q_2}\left[\frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}}\right]^T \times \frac{\partial F}{\partial \mathbf{a}(t)} \tag{3.72}$$

If we solve the second term in the summation of the previous equation for the interval *1* to *Q-N*:

$$\begin{aligned}\frac{\partial F}{\partial \mathbf{a}(1)} &= \frac{\partial^e \mathbf{a}(2)}{\partial \mathbf{a}(1)} \times \frac{\partial F}{\partial \mathbf{a}(2)} \\ \frac{\partial F}{\partial \mathbf{a}(2)} &= \frac{\partial^e \mathbf{a}(3)}{\partial \mathbf{a}(2)} \times \frac{\partial F}{\partial \mathbf{a}(3)} \\ \\ \frac{\partial F}{\partial \mathbf{a}(Q-N)} &= \frac{\partial^e \mathbf{a}(Q-N+1)}{\partial \mathbf{a}(Q-N)} \times \frac{\partial F}{\partial \mathbf{a}(Q-N+1)}\end{aligned} \tag{3.73}$$

we will obtain for a generic time *t=k*:

$$\frac{\partial F}{\partial \mathbf{a}(k)} = \frac{\partial^e \mathbf{a}(k+1)}{\partial \mathbf{a}(k)} \times \frac{\partial^e \mathbf{a}(k+2)}{\partial \mathbf{a}(k+1)} \times \ldots \times \frac{\partial^e \mathbf{a}(Q-N+1)}{\partial \mathbf{a}(Q-N)} \times \frac{\partial F}{\partial \mathbf{a}(Q-N+1)} . \tag{3.74}$$

This result implies that:

$$Z(1, Q-N) = \sum_{t=1}^{Q-N} \left[\frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}}\right]^T \times \frac{\partial^e \mathbf{a}(t+1)}{\partial \mathbf{a}(t)} \times \frac{\partial^e \mathbf{a}(t+2)}{\partial \mathbf{a}(t+1)} \times \ldots \times \frac{\partial^e \mathbf{a}(Q-N+1)}{\partial \mathbf{a}(Q-N)} \times \frac{\partial F}{\partial \mathbf{a}(Q-N+1)} \tag{3.75}$$

If we define:

$$\mathbf{Q}_i(Q-N) = \sum_{t=1}^{Q-N} \left[\frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}}\right]^T \times \frac{\partial^e \mathbf{a}(t+1)}{\partial \mathbf{a}(t)} \times \frac{\partial^e \mathbf{a}(t+2)}{\partial \mathbf{a}(t+1)} \times \ldots \times \frac{\partial^e \mathbf{a}(Q-N+1)}{\partial \mathbf{a}(Q-N)} \tag{3.76}$$

we have:

$$Z(1, Q-N) = \mathbf{Q}_i(Q-N) \times \frac{\partial F}{\partial \mathbf{a}(Q-N+1)} . \tag{3.77}$$

If we apply Eq. (3.54) for the interval *t=Q-N+1* to *t=Q*:

$$\begin{aligned}
\frac{\partial F}{\partial \mathbf{a}(Q)} &= \frac{\partial^e F}{\partial \mathbf{a}(Q)} + \cancel{\frac{\partial^e \mathbf{a}(Q+1)}{\partial \mathbf{a}(Q)} \times \frac{\partial F}{\partial \mathbf{a}(Q+1)}} \\
\frac{\partial F}{\partial \mathbf{a}(Q-1)} &= \frac{\partial^e F}{\partial \mathbf{a}(Q-1)} + \frac{\partial^e \mathbf{a}(Q)}{\partial \mathbf{a}(Q-1)} \times \frac{\partial F}{\partial \mathbf{a}(Q)} \\
&\ldots \\
\frac{\partial F}{\partial \mathbf{a}(Q-N+1)} &= \frac{\partial^e F}{\partial \mathbf{a}(Q-N+1)} + \frac{\partial^e \mathbf{a}(Q-N+2)}{\partial \mathbf{a}(Q-N+1)} \times \frac{\partial F}{\partial \mathbf{a}(Q-N+2)}
\end{aligned} \tag{3.78}$$

we will obtain for a generic time *t=k*:

$$\begin{aligned}
\frac{\partial F}{\partial \mathbf{a}(k)} &= \frac{\partial^e F}{\partial \mathbf{a}(k)} + \frac{\partial^e \mathbf{a}(k+1)}{\partial \mathbf{a}(k)} \times \left(\frac{\partial^e F}{\partial \mathbf{a}(k+1)} + \frac{\partial^e \mathbf{a}(k+2)}{\partial \mathbf{a}(k+1)} \times (\ldots\right. \\
&\quad \left.\ldots \times \left(\frac{\partial^e F}{\partial \mathbf{a}(Q-1)} + \frac{\partial^e \mathbf{a}(Q)}{\partial \mathbf{a}(Q-1)} \times \frac{\partial^e F}{\partial \mathbf{a}(Q)}\right) \ldots\right) \\
&= \frac{\partial^e F}{\partial \mathbf{a}(k)} + \frac{\partial^e \mathbf{a}(k+1)}{\partial \mathbf{a}(k)} \times \frac{\partial^e F}{\partial \mathbf{a}(k+1)} + \frac{\partial^e \mathbf{a}(k+1)}{\partial \mathbf{a}(k)} \times \frac{\partial^e \mathbf{a}(k+2)}{\partial \mathbf{a}(k+1)} \times \frac{\partial^e F}{\partial \mathbf{a}(k+2)} + \ldots \\
&\quad + \frac{\partial^e \mathbf{a}(k+1)}{\partial \mathbf{a}(k)} \times \frac{\partial^e \mathbf{a}(k+2)}{\partial \mathbf{a}(k+1)} \times \ldots \times \frac{\partial^e \mathbf{a}(Q)}{\partial \mathbf{a}(Q-1)} \times \frac{\partial^e F}{\partial \mathbf{a}(Q)}
\end{aligned} \tag{3.79}$$

We can rewrite Eq. (3.71) as:

$$\begin{aligned} G(Q) &= Z(1, Q-N) + Z(Q-N+1, Q) \\ &= \mathbf{Q}_i(Q-N) \times \frac{\partial F}{\partial \mathbf{a}(Q-N+1)} + \sum_{t=Q-N+1}^{Q} \left[\frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}}\right]^T \times \left(\frac{\partial^e F}{\partial \mathbf{a}(k)} + \frac{\partial^e \mathbf{a}(k+1)}{\partial \mathbf{a}(k)} \times \frac{\partial^e F}{\partial \mathbf{a}(k+1)}\right. \\ &\quad + \frac{\partial^e \mathbf{a}(k+1)}{\partial \mathbf{a}(k)} \times \frac{\partial^e \mathbf{a}(k+2)}{\partial \mathbf{a}(k+1)} \times \frac{\partial^e F}{\partial \mathbf{a}(k+2)} + \ldots \\ &\quad \left.\ldots + \frac{\partial^e \mathbf{a}(k+1)}{\partial \mathbf{a}(k)} \times \frac{\partial^e \mathbf{a}(k+2)}{\partial \mathbf{a}(k+1)} \times \ldots \times \frac{\partial^e \mathbf{a}(Q)}{\partial \mathbf{a}(Q-1)} \times \frac{\partial^e F}{\partial \mathbf{a}(Q)}\right) \end{aligned} \quad (3.80)$$

where:

$$\begin{aligned} \frac{\partial F}{\partial \mathbf{a}(Q-N+1)} &= \frac{\partial^e F}{\partial \mathbf{a}(Q-N+1)} + \frac{\partial^e \mathbf{a}(Q-N+2)}{\partial \mathbf{a}(Q-N+1)} \times \frac{\partial^e F}{\partial \mathbf{a}(Q-N+2)} + \ldots \\ &\quad + \frac{\partial^e \mathbf{a}(Q-N+2)}{\partial \mathbf{a}(Q-N+1)} \times \frac{\partial^e \mathbf{a}(Q-N+3)}{\partial \mathbf{a}(Q-N+2)} \times \ldots \times \frac{\partial^e \mathbf{a}(Q)}{\partial \mathbf{a}(Q-1)} \times \frac{\partial^e F}{\partial \mathbf{a}(Q)} \end{aligned} . \quad (3.81)$$

We could define a new term:

$$\begin{aligned} \Gamma(t, Q) &= \frac{\partial^e F}{\partial \mathbf{a}(k)} + \frac{\partial^e \mathbf{a}(k+1)}{\partial \mathbf{a}(k)} \times \frac{\partial^e F}{\partial \mathbf{a}(k+1)} + \frac{\partial^e \mathbf{a}(k+1)}{\partial \mathbf{a}(k)} \times \frac{\partial^e \mathbf{a}(k+2)}{\partial \mathbf{a}(k+1)} \times \frac{\partial^e F}{\partial \mathbf{a}(k+2)} + \ldots \\ &\quad \ldots + \frac{\partial^e \mathbf{a}(k+1)}{\partial \mathbf{a}(k)} \times \frac{\partial^e \mathbf{a}(k+2)}{\partial \mathbf{a}(k+1)} \times \ldots \times \frac{\partial^e \mathbf{a}(Q)}{\partial \mathbf{a}(Q-1)} \times \frac{\partial^e F}{\partial \mathbf{a}(Q)} \end{aligned} \quad (3.82)$$

that will allow us to rewrite Eq. (3.80) as:

$$G(Q) = \sum_{t=Q-N+1}^{Q} \left[\frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}}\right]^T \times \Gamma(t, Q) + \mathbf{Q}_i(Q-N) \times \Gamma(Q-N+1, Q) \quad (3.83)$$

where we can calculate recursively the term in Eq. (3.82) with:

$$\Gamma(t, Q) = \frac{\partial^e F}{\partial \mathbf{a}(k)} + \frac{\partial^e \mathbf{a}(k+1)}{\partial \mathbf{a}(k)} \times \Gamma(t+1, Q) \quad (3.84)$$

Atiya and Parlos [2] demonstrated that the Forward Perturbation Method has the largest computational complexity $O(N^4)$, followed by Green's Function, Fast-Forward Propagation and Block Update Methods with computational complexity $O(N^3)$. The Backpropagation-Through-Time method has the smallest computational complexity $O(N^2)$. Here N is the number of nodes in a fully recurrent network of the form

$$x(k+1) = f[Wx(k)] \qquad k = 0, \ldots, K-1 \quad (3.85)$$

where k is the time index.

In section 8.3 we will review the computational complexity of the Forward Perturbation and Backpropagation-Through-Time Methods, based not only on the gradient calculation but also on the Jacobian calculation. After reviewing the test networks presented on Appendix D, we concluded that it is not appropriate to use the number of nodes in the network to determine computational complexity when using a variety of different network architectures. A better approach is to consider the total number of weights in the network. Therefore, for the computational complexity mentioned above, the Forward Perturbation Method has computational complexity $O(N^2)$ and the Backpropagation-Through-Time has computational complexity $O(N)$, where N is the number of weights and biases in the network.

Appendix B describes how all of the methods described previously can be applied to a single layer neural network with one delay. Table 3.1 shows the floating point operations for that network. BTT is the method with the fewest floating points operations, followed by FP. The Fast Forward Propagation algorithm results in more than double the number of operations of the BTT or the FP methods. The Green's Function method also required more flops than BTT and FP. Those two methods are the only ones that required matrix inversion, producing ill-conditioning problems. Because these methods did not improve the number of flops, we will not consider them further in this dissertation. The Block Update method requires more flops than the BTT and the FP methods. However, this method could be useful for long sequences, where memory requirements could jeopardize the algorithm implementation.

Table 3.1: *Floating point operations for dynamic gradient calculations of Appendix B example.*

Method	Additions	Multiplications	Divisions	Flops
Forward Perturbation	10	7	0	17
Backpropagation-Through-Time	6	7	0	13
Green's Function	10	13	4	27
Fast Forward Propagation	14	21	3	38
Block Update	10	10	0	20

3.5. Summary.

In this chapter we have introduced the key concepts required for training recurrent networks. In particular, we have discussed two important algorithms for computing gradients for dynamic networks: the Forward Perturbation algorithm and Backpropagation Through Time. In the next two chapters we will generalize these two algorithms for the computation of gradients for arbitrary LDDNs, and we will extend the algorithms for Jacobian calculations as well.

Chapter 4

FORWARD PERTURBATION

In this section we will generalize the Forward Perturbation (FP) algorithm, given in Eq. (3.51) and Eq. (3.53), for LDDN networks [9]. We will begin with some preliminary definitions.

4.1. Preliminaries

To explain the algorithms, we must create certain definitions related to the LDDN. We do that in the following paragraphs.

First, as we stated earlier, a *layer* consists of a set of *weights*, associated *tapped delay lines*, a *summing junction*, and a *transfer function*. The network has *inputs* that are connected to special weights, called *input weights*, and denoted by $\mathbf{IW}^{i,j}$, where j denotes the number of the input vector that enters the weight, and i denotes the number of the layer to which the weight is connected. The weights connecting one layer to another are called *layer weights* and are denoted by $\mathbf{LW}^{i,j}$, where j denotes the number of the layer coming into the weight and i denotes the number of the layer at the output of weight. In order to calculate the network response in stages, layer by layer, we need to proceed in the proper layer order, so that the necessary inputs at each layer will be available. This ordering of layers is called the *simulation order*. In order to backpropagate the derivatives for the gradient calculations, we must proceed in the opposite order, which is called the *backpropagation order*.

In order to simplify the description of the training algorithm, some layers of the LDDN will be assigned as network outputs, and some will be assigned as network inputs. A layer is an *input layer* if it has an input weight, or if it contains any delays with any of its weight matrices. A layer is an *output layer* if its output will be compared to a target during training, or if it is connected to an input layer through a matrix which has any delays associated with it.

For example, the LDDN shown in Figure 4.1 has two output layers (1 and 3) and two input layers (1 and 2). For this network the simulation order is 1-2-3, and the backpropagation order is 3-2-1. As an aid in later derivations, we will define U as the set of all output layer numbers and X as the set of all input layer numbers. For the LDDN in Figure 4.1, U={1,3} and X={1,2}.

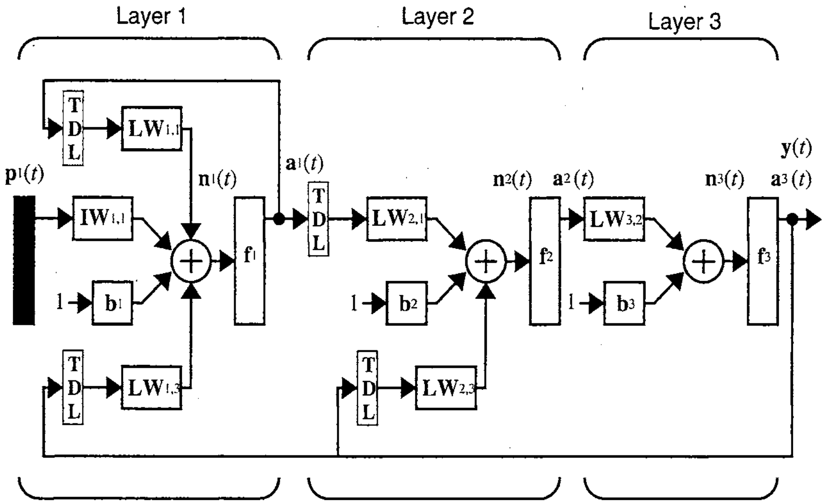

Figure 4.1: *Three-layer LDDN with two output layers (1 and 3) and two input layers (1 and 2)*

The general equations for simulating an arbitrary LDDN network are given below. The net input at layer m can be computed as

$$\mathbf{n}^m(t) = \sum_{l \in L_m^f} \sum_{d \in DL_{m,l}} \mathbf{LW}^{m,l}(d)\mathbf{a}^l(t-d) + \sum_{l \in I_m} \sum_{d \in DI_{m,l}} \mathbf{IW}^{m,l}(d)\mathbf{p}^l(t-d) + \mathbf{b}^m \qquad \textbf{(4.1)}$$

where $DL_{m,l}$ is the set of all delays in the tapped delay line between Layer l and Layer m, $DI_{m,l}$ is the set of all delays in the tapped delay line between Input l and Layer m, I_m is the set of indices of input vectors that connect forward to layer m, and L_m^f is the set of indices of layers that directly connect *forward* to layer m. The output of layer m is then computed as

$$\mathbf{a}^m(t) = \mathbf{f}^m(\mathbf{n}^m(t)) . \qquad \textbf{(4.2)}$$

At each time point, Eq. (4.1) and Eq. (4.2) are iterated forward through the layers, as m is incremented through the simulation order. Time is then incremented from t=1 to t=Q.

4.2. Eq. (3.51)

The first step in generalizing the FP algorithm is to generalize Eq. (3.51). For the general LDDN network, we can calculate the terms of the gradient by using the chain rule, as in

$$\frac{\partial F}{\partial w} = \sum_{t=1}^{Q} \sum_{u \in U} \left\{ \left[\frac{\partial \mathbf{a}^{u}(t)}{\partial w} \right]^{T} \times \frac{\partial^{e} F}{\partial \mathbf{a}^{u}(t)} \right\}, \tag{4.3}$$

where u is an output layer, U is the set of all output layer numbers, and w represents $lw_{i,j}^{m,l}(d)$, $iw_{i,j}^{m,l}(d)$ and b_i^m . (The equation is the same for all network parameters.)

4.3. Eq. (3.53)

The next step of the development of the FP algorithm is the generalization of Eq. (3.53). Again, we use the chain rule:

$$\frac{\partial \mathbf{a}^{u}(t)}{\partial w} = \frac{\partial^{e} \mathbf{a}^{u}(t)}{\partial w} + \sum_{u' \in U} \sum_{x \in X} \sum_{d \in DL_{x,u'}} \frac{\partial^{e} \mathbf{a}^{u}(t)}{\partial \mathbf{n}^{x}(t)^{T}} \times \frac{\partial^{e} \mathbf{n}^{x}(t)}{\partial \mathbf{a}^{u'}(t-d)^{T}} \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial w} \tag{4.4}$$

In Eq. (3.53) we only had one delay in the system. Now we need to account for each output and also for the number of times each output is delayed before it is input to another layer. That is the reason for the summations in Eq. (4.4). These equations must be updated forward in time, as t is varied from 1 to Q. The terms

$$\frac{\partial \mathbf{a}^{u}(t)}{\partial w} \tag{4.5}$$

are generally set to zero for $t \le 0$.

To implement Eq. (4.4) we need to compute the terms

$$\frac{\partial^{e} \mathbf{a}^{u}(t)}{\partial \mathbf{n}^{x}(t)^{T}} \times \frac{\partial^{e} \mathbf{n}^{x}(t)}{\partial \mathbf{a}^{u'}(t-d)^{T}} \tag{4.6}$$

To find the second term on the right, we can use

$$n_k^x(t) = \sum_{l \in L_x^f} \sum_{d' \in DL_{x,l}} \left(\sum_{i=1}^{S_l} lw_{k,i}^{x,l}(d') a_i^l(t-d') \right) + \sum_{l \in L_x^f} \sum_{d' \in DI_{x,l}} \left(\sum_{i=1}^{S_l} iw_{k,i}^{x,l}(d') p_i^l(t-d') \right) + b_k^x \tag{4.7}$$

we can now write

$$\frac{\partial^e n_k^x(t)}{\partial a_j^{u'}(t-d)} = lw_{k,j}^{x,u'}(d) \tag{4.8}$$

If we define the following sensitivity term

$$s_{k,i}^{u,m}(t) \equiv \frac{\partial^e a_k^u(t)}{\partial n_i^m(t)}, \tag{4.9}$$

which can be used to make up the following matrix

$$\begin{aligned} \mathbf{S}^{u,m}(t) &= \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{n}^m(t)^T} = \begin{bmatrix} s_{1,1}^{u,m}(t) & s_{1,2}^{u,m}(t) & \ldots & s_{1,S_m}^{u,m}(t) \\ s_{2,1}^{u,m}(t) & s_{2,2}^{u,m}(t) & \ldots & s_{1,S_m}^{u,m}(t) \\ \\ s_{S_u,1}^{u,m}(t) & s_{S_u,2}^{u,m}(t) & \ldots & s_{S_u,S_m}^{u,m}(t) \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{s}_1^{u,m}(t) & \mathbf{s}_2^{u,m}(t) & \ldots & \mathbf{s}_{S_m}^{u,m}(t) \end{bmatrix} = \begin{bmatrix} {}_1\mathbf{s}^{u,m}(t)^T \\ {}_2\mathbf{s}^{u,m}(t)^T \\ \\ {}_{S_u}\mathbf{s}^{u,m}(t)^T \end{bmatrix} \end{aligned} \tag{4.10}$$

then we can write Eq. (4.6) as

$$\left[\frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{n}^x(t)^T} \times \frac{\partial^e \mathbf{n}^x(t)}{\partial \mathbf{a}^{u'}(t-d)^T}\right]_{i,j} = \sum_{k=1}^{S_x} s_{i,k}^{u,x}(t+d) \times lw_{k,j}^{x,u'}(d) \tag{4.11}$$

or in matrix form

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{n}^x(t)^T} \times \frac{\partial^e \mathbf{n}^x(t)}{\partial \mathbf{a}^{u'}(t-d)^T} = \mathbf{S}^{u,x}(t) \times \mathbf{LW}^{x,u'}(d). \tag{4.12}$$

and therefore Eq. (4.4) can be written

$$\frac{\partial \mathbf{a}^u(t)}{\partial w} = \frac{\partial^e \mathbf{a}^u(t)}{\partial w} + \sum_{u' \in U} \sum_{x \in X} \sum_{d \in DL_{x,u'}} \mathbf{S}^{u,x}(t) \times \mathbf{LW}^{x,u'}(d) \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial w} \tag{4.13}$$

Many of the terms in the summation on the right hand side of Eq. (4.13) will be zero and will not have to be computed. To take advantage of these efficiencies, we introduce the following definitions.

$$E^{U}_{LW}(x) = \{u \in U \ni \exists(\mathbf{LW}^{x,\,u} \neq 0)\} \tag{4.14}$$

$$E^{X}_{S}(u) = \{x \in X \ni \exists(\mathbf{S}^{u,\,x} \neq 0)\} \tag{4.15}$$

$$E_{S}(u) = \{x \ni \exists(\mathbf{S}^{u,\,x} \neq 0)\} \tag{4.16}$$

Using Eq. (4.14) and Eq. (4.15), we can rearrange the order of the summations in Eq. (4.13) and sum only over existing terms

$$\frac{\partial \mathbf{a}^{u}(t)}{\partial w} = \frac{\partial^{e} \mathbf{a}^{u}(t)}{\partial w} + \sum_{x \in E^{X}_{S}(u)} \mathbf{S}^{u,\,x}(t) \sum_{u' \in E^{U}_{LW}(x)} \sum_{d \in DL_{x,\,u'}} \mathbf{LW}^{x,\,u'}(d) \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial w} \tag{4.17}$$

This can be written for the individual weight matrices

$$\begin{aligned} \frac{\partial \mathbf{a}^{u}(t)}{\partial vec(\mathbf{LW}^{m,\,l}(d))^{T}} &= \frac{\partial^{e} \mathbf{a}^{u}(t)}{\partial vec(\mathbf{LW}^{m,\,l}(d))^{T}} \\ &\quad + \sum_{x \in E^{X}_{S}(u)} \mathbf{S}^{u,\,x}(t) \sum_{u' \in E^{U}_{LW}(x)} \sum_{d \in DL_{x,\,u'}} \mathbf{LW}^{x,\,u'}(d) \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial vec(\mathbf{LW}^{m,\,l}(d))^{T}} \end{aligned} \tag{4.18}$$

$$\begin{aligned} \frac{\partial \mathbf{a}^{u}(t)}{\partial vec(\mathbf{IW}^{m,\,l}(d))^{T}} &= \frac{\partial^{e} \mathbf{a}^{u}(t)}{\partial vec(\mathbf{IW}^{m,\,l}(d))^{T}} \\ &\quad + \sum_{x \in E^{X}_{S}(u)} \mathbf{S}^{u,\,x}(t) \sum_{u' \in E^{U}_{LW}(x)} \sum_{d \in DL_{x,\,u'}} \mathbf{LW}^{x,\,u'}(d) \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial vec(\mathbf{IW}^{m,\,l}(d))^{T}} \end{aligned} \tag{4.19}$$

$$\frac{\partial \mathbf{a}^{u}(t)}{\partial (\mathbf{b}^{m})^{T}} = \frac{\partial^{e} \mathbf{a}^{u}(t)}{\partial (\mathbf{b}^{m})^{T}} + \sum_{x \in E^{X}_{S}(u)} \mathbf{S}^{u,\,x}(t) \sum_{u' \in E^{U}_{LW}(x)} \sum_{d \in DL_{x,\,u'}} \mathbf{LW}^{x,\,u'}(d) \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial (\mathbf{b}^{m})^{T}} \tag{4.20}$$

where the *vec* operator transforms a matrix into a vector by stacking the columns of the matrix one underneath the other [29].

Eq. (4.17) through Eq. (4.20) make up the generalization of Eq. (3.53) for the LDDN network. It remains to compute the sensitivity matrices $\mathbf{S}^{u,\,m}(t)$ and the explicit derivatives $\partial^{e} \mathbf{a}^{u}(t)/\partial w$, which are described in the next two sections.

4.4. Sensitivities

In order to compute the elements of the sensitivity matrix, we use a form of backpropagation. The sensitivities at the outputs of the network can be computed as

$$s_{k,i}^{u,u}(t) = \frac{\partial^e a_k^u(t)}{\partial n_i^u(t)} = \begin{cases} \dot{f}^u(n_i^u(t)) & \text{for } i = k \\ 0 & \text{for } i \neq k \end{cases}, u \in U, \tag{4.21}$$

or, in matrix form,

$$\mathbf{S}^{u,u}(t) = \dot{\mathbf{F}}^u(\mathbf{n}^u(t)), \tag{4.22}$$

where $\dot{\mathbf{F}}^u(\mathbf{n}^u(t))$ is defined as

$$\dot{\mathbf{F}}^u(\mathbf{n}^u(t)) = \begin{bmatrix} \dot{f}^u(n_1^u(t)) & 0 & \dots & 0 \\ 0 & \dot{f}^u(n_2^u(t)) & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & \dot{f}^u(n_{S^u}^u(t)) \end{bmatrix} \tag{4.23}$$

The matrices $\mathbf{S}^{u,m}(t)$ can be computed by backpropagating through the network, from each network output, using

$$\mathbf{S}^{u,m}(t) = \left\{ \sum_{l \in L_m^b} \mathbf{S}^{u,l}(t)\mathbf{LW}^{l,m}(0) \right\} \dot{\mathbf{F}}^m(\mathbf{n}^m(t)), u \in U, \tag{4.24}$$

where m is decremented from u through the backpropagation order and L_m^b is the set of indices of layers that are directly connected backwards to layer m (or to which layer m connects forward) and that contain no delays.

4.5. Explicit Derivatives

We also need to compute the explicit derivatives

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial w}. \tag{4.25}$$

We can derive the following three expansions of Eq. (4.25):

$$\frac{\partial^e a_k^u(t)}{\partial iw_{i,j}^{m,l}(d)} = \frac{\partial^e a_k^u(t)}{\partial n_i^m(t)} \times \frac{\partial^e n_i^m(t)}{\partial iw_{i,j}^{m,l}(d)} = s_{i,k}^{u,m}(t) \times p_j^l(t-d), \tag{4.26}$$

$$\frac{\partial^e a_k^u(t)}{\partial lw_{i,j}^{m,l}(d)} = \frac{\partial^e a_k^u(t)}{\partial n_i^m(t)} \times \frac{\partial^e n_i^m(t)}{\partial lw_{i,j}^{m,l}(d)} = s_{i,k}^{u,m}(t) \times a_j^l(t-d), \tag{4.27}$$

$$\frac{\partial^e a_k^u(t)}{\partial b_i^m} = \frac{\partial^e a_k^u(t)}{\partial n_i^m(t)} \times \frac{\partial^e n_i^m(t)}{\partial b_i^m} = s_{i,k}^{u,m}(t). \tag{4.28}$$

In vector form we can write

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial iw_{i,j}^{m,l}(d)} = \mathbf{s}_i^{u,m}(t) \times p_j^l(t-d), \tag{4.29}$$

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial lw_{i,j}^{m,l}(d)} = \mathbf{s}_i^{u,m}(t) \times a_j^l(t-d), \tag{4.30}$$

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial b_i^m} = \mathbf{s}_i^{u,m}(t). \tag{4.31}$$

In matrix form we have

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial vec(\mathbf{IW}^{m,l}(d))^T} = [\mathbf{p}^l(t-d)]^T \otimes \mathbf{S}^{u,m}(t), \tag{4.32}$$

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial vec(\mathbf{LW}^{m,l}(d))^T} = [\mathbf{a}^l(t-d)]^T \otimes \mathbf{S}^{u,m}(t), \tag{4.33}$$

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial (\mathbf{b}^m)^T} = \mathbf{S}^{u,m}(t), \tag{4.34}$$

where $\mathbf{A} \otimes \mathbf{B}$ is the Kronecker product of $\mathbf{A}$ and $\mathbf{B}$ [29].

4.6. FP Gradient Algorithm Summary.

The total FP gradient calculation algorithm for the LDDN network is summarized in Figure 4.2.

```
Forward Perturbation Gradient
Initialize:
    $\frac{\partial \mathbf{a}^u(t)}{\partial w} = 0, t \le 0$, for all $u \in U$,
For $t = 1$ to $Q$,
    $U' = \varnothing$, $E_S(u) = \varnothing$ and $E_S^X(u) = \varnothing$ for all $u \in U$.
    For $m$ decremented through the BP order
        For all $u \in U'$
            $\mathbf{S}^{u,m}(t) = \left\{ \sum_{l \in E_S(u) \cap L_m^b} \mathbf{S}^{u,l}(t)\mathbf{LW}^{l,m}(0) \right\} \dot{\mathbf{F}}^m(\mathbf{n}^m(t))$
            add $m$ to the set $E_S(u)$
            if $m \in X$, add $m$ to the set $E_S^X(u)$
        EndFor $u$
        If $m \in U$
            $\mathbf{S}^{m,m}(t) = \dot{\mathbf{F}}^m(\mathbf{n}^m(t))$
            add $m$ to the sets $U'$ and $E_S(m)$
            if $m \in X$, add $m$ to the set $E_S^X(m)$
        EndIf $m$
    EndFor $m$
    For $u \in U$ incremented through the simulation order
        For all weights and biases (w is a vector containing all weights and biases)
            $\frac{\partial^e \mathbf{a}^u(t)}{\partial vec(\mathbf{IW}^{m,l}(d))^T} = [\mathbf{p}^l(t-d)]^T \otimes \mathbf{S}^{u,m}(t)$
            $\frac{\partial^e \mathbf{a}^u(t)}{\partial vec(\mathbf{LW}^{m,l}(d))^T} = [\mathbf{a}^l(t-d)]^T \otimes \mathbf{S}^{u,m}(t)$
            $\frac{\partial^e \mathbf{a}^u(t)}{\partial(\mathbf{b}^m)^T} = \mathbf{S}^{u,m}(t)$
        EndFor weights and biases
        $\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{w}^T} = \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{w}^T} + \sum_{x \in E_S^X(u)} \mathbf{S}^{u,x}(t) \sum_{u' \in E_{LW}^U(x)} \sum_{d \in DL_{x,u'}} \mathbf{LW}^{x,u'}(d) \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial \mathbf{w}^T}$
    EndFor $u$
EndFor $t$
Compute gradients
    $\frac{\partial F}{\partial \mathbf{w}^T} = \sum_{t=1}^{Q} \sum_{u \in U} \left\{ \left[ \frac{\partial^e F}{\partial \mathbf{a}^u(t)} \right]^T \times \frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{w}^T} \right\}$
```

Figure 4.2: *Pseudo Code for the Forward Perturbation gradient algorithm.*

4.7. FP Jacobian Algorithm.

The previous section described the forward perturbation algorithm for computing the gradient. This gradient could be used for steepest descent, conjugate gradient and quasi-Newton training algorithms as described from section 3.1.1 to section 3.1.3. If we want to implement a Gauss-Newton or a Levenberg-Marquardt algorithm, we need to calculate the Jacobian instead of the gradient. In this section we will present the FP implementation of the Jacobian calculation.

The weight update process using the Levenberg-Marquardt Algorithm was presented in Eq. (3.38) and Eq. (3.39). If we rewrite the last equation using the terminology given in this chapter:

$$\Delta\mathbf{w}_p = -[\mathbf{J}^T(\mathbf{w}_p)\mathbf{J}(\mathbf{w}_p) + \mu_p\mathbf{I}]^{-1}\mathbf{J}^T(\mathbf{w}_p)\mathbf{v}(\mathbf{w}_p) \tag{4.35}$$

where $\mathbf{w}_p$ is a vector containing all weights and biases in the network.

We need the Jacobian matrix $\mathbf{J}(\mathbf{w}_p)$ to perform the weight update for a given epoch or iteration p. We can express the Jacobian using Eq. (3.30) as:

$$\mathbf{J}^T(\mathbf{w}) = \begin{bmatrix} \frac{\partial v_1(1)}{\partial w_1} & \frac{\partial v_1(1)}{\partial w_2} & \dots & \frac{\partial v_1(1)}{\partial w_n} \\ \frac{\partial v_2(1)}{\partial w_1} & \frac{\partial v_2(1)}{\partial w_2} & \dots & \frac{\partial v_2(1)}{\partial w_n} \\ \dots & \dots & & \dots \\ \frac{\partial v_N(1)}{\partial w_1} & \frac{\partial v_N(1)}{\partial w_2} & \dots & \frac{\partial v_N(1)}{\partial w_n} \\ \frac{\partial v_1(2)}{\partial w_1} & \frac{\partial v_1(2)}{\partial w_2} & \dots & \frac{\partial v_1(2)}{\partial w_n} \\ \dots & \dots & \dots & \dots \\ \frac{\partial v_N(Q-1)}{\partial w_1} & \frac{\partial v_N(Q-1)}{\partial w_2} & \dots & \frac{\partial v_N(Q-1)}{\partial w_n} \\ \frac{\partial v_1(Q)}{\partial w_1} & \frac{\partial v_1(Q)}{\partial w_2} & \dots & \frac{\partial v_1(Q)}{\partial w_n} \\ \dots & \dots & \dots & \dots \\ \frac{\partial v_N(Q)}{\partial w_1} & \frac{\partial v_N(Q)}{\partial w_2} & \dots & \frac{\partial v_N(Q)}{\partial w_n} \end{bmatrix} \tag{4.36}$$

where $v_k(t)$ represents the individual error function for an individual output element k at time t:

$$v_k(t) = t_k(t) - a_k(t) \tag{4.37}$$

where $t_k(t)$ is the target and $a_k(t)$ is the network output at time t. The index k represents the individual target or output for all the layers contained in the set of output layers U arranged in one vector from 1 to N

$$\mathbf{a} = \begin{bmatrix} \mathbf{a}^{u_1} \\ \mathbf{a}^{u_2} \\ \dots \\ \mathbf{a}^{u_{Nu}} \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \dots \\ a_{N-1} \\ a_N \end{bmatrix} \quad ; \quad \mathbf{t} = \begin{bmatrix} \mathbf{t}^{u_1} \\ \mathbf{t}^{u_2} \\ \dots \\ \mathbf{t}^{u_{Nu}} \end{bmatrix} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ \dots \\ t_{N-1} \\ t_N \end{bmatrix},$$

where N_u is the number of output layers and $N = \sum_{u \in U} S_u$.

The k, j element of the Jacobian is

$$\frac{\partial v_k(t)}{\partial w_j} = \frac{\partial(t_k(t) - a_k(t))}{\partial w_j} = -\frac{\partial a_k(t)}{\partial w_j} \tag{4.38}$$

This is the negative of the dynamic derivative of the network output, which is computed as part of the FP calculation of the gradient (Eq. (4.32) to Eq. (4.34)). As also noticed by Yang $^{(55)}$, the Jacobian generation does not require the calculation of the derivative of the objective function, as in Eq. (4.3). Therefore, the FP implementation of the Jacobian calculation is actually a simplified version of the FP gradient algorithm.

4.8. FP Jacobian Algorithm Summary.

The total FP Jacobian algorithm for the LDDN network is summarized in Figure 4.3.

```
Initialize:                                                        Forward Perturbation Jacobian
    $\frac{\partial \mathbf{a}^u(t)}{\partial w} = 0, t \le 0$, for all $u \in U$,
For $t = 1$ to $Q$,
    $U' = \varnothing$, $E_S(u) = \varnothing$ and $E_S^X(u) = \varnothing$ for all $u \in U$.
    For $m$ decremented through the BP order
        For all $u \in U'$
            $\mathbf{S}^{u,m}(t) = \left\{ \sum_{l \in E_S(u) \cap L_m^b} \mathbf{S}^{u,l}(t)\mathbf{LW}^{l,m}(0) \right\} \dot{\mathbf{F}}^m(\mathbf{n}^m(t))$
            add $m$ to the set $E_S(u)$
            if $m \in X$, add $m$ to the set $E_S^X(u)$
        EndFor $u$
        If $m \in U$
            $\mathbf{S}^{m,m}(t) = \dot{\mathbf{F}}^m(\mathbf{n}^m(t))$
            add $m$ to the sets $U'$ and $E_S(m)$
            if $m \in X$, add $m$ to the set $E_S^X(m)$
        EndIf $m$
    EndFor $m$
    For $u \in U$ incremented through the simulation order
        For all weights and biases (w is a vector containing all weights and biases)
            $\frac{\partial^e \mathbf{a}^u(t)}{\partial vec(\mathbf{IW}^{m,l}(d))^T} = [\mathbf{p}^l(t-d)]^T \otimes \mathbf{S}^{u,m}(t)$
            $\frac{\partial^e \mathbf{a}^u(t)}{\partial vec(\mathbf{LW}^{m,l}(d))^T} = [\mathbf{a}^l(t-d)]^T \otimes \mathbf{S}^{u,m}(t)$
            $\frac{\partial^e \mathbf{a}^u(t)}{\partial(\mathbf{b}^m)^T} = \mathbf{S}^{u,m}(t)$
        EndFor weights and biases
        $\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{w}^T} = \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{w}^T} + \sum_{x \in E_S^X(u)} \mathbf{S}^{u,x}(t) \sum_{u' \in E_{LW}^U(x)} \sum_{d \in DL_{x,u'}} \mathbf{LW}^{x,u'}(d) \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial \mathbf{w}^T}$
    EndFor $u$
EndFor $t$
```

Figure 4.3: *Pseudo Code for the Forward Perturbation Jacobian algorithm.*

Chapter 5

BACKPROPAGATION THROUGH TIME

In this section we will generalize the Backpropagation-Through-Time (BTT) algorithm, given in Eq. (3.52) and Eq. (3.54), for LDDN networks [8].

5.1. Eq. (3.52)

The first step is to generalize Eq. (3.52). For the general LDDN network, we can calculate the terms of the gradient by using the chain rule, as in

$$\frac{\partial F}{\partial lw_{i,j}^{m,l}(d)} = \sum_{t=1}^{Q}\left\{\sum_{u \in U}\sum_{k=1}^{S_u}\frac{\partial F}{\partial a_k^u(t)} \times \frac{\partial^e a_k^u(t)}{\partial n_i^m(t)}\right\}\frac{\partial^e n_i^m(t)}{\partial lw_{i,j}^{m,l}(d)} \tag{5.1}$$

$$\frac{\partial F}{\partial iw_{i,j}^{m,l}(d)} = \sum_{t=1}^{Q}\left\{\sum_{u \in U}\sum_{k=1}^{S_u}\frac{\partial F}{\partial a_k^u(t)} \times \frac{\partial^e a_k^u(t)}{\partial n_i^m(t)}\right\}\frac{\partial^e n_i^m(t)}{\partial iw_{i,j}^{m,l}(d)} \tag{5.2}$$

$$\frac{\partial F}{\partial b_i^m} = \sum_{t=1}^{Q}\left\{\sum_{u \in U}\sum_{k=1}^{S_u}\frac{\partial F}{\partial a_k^u(t)} \times \frac{\partial^e a_k^u(t)}{\partial n_i^m(t)}\right\}\frac{\partial^e n_i^m(t)}{\partial b_i^m} \tag{5.3}$$

where u is an output layer, U is the set of all output layers, and S_u is the number of neurons in layer u.

From Eq. (4.1), the elements of the net input can be computed as

$$n_i^m(t) = \sum_{l \in L_m^f}\sum_{d \in DL_{m,l}}\left(\sum_{j=1}^{S_l} lw_{i,j}^{m,l}(d)a_j^l(t-d)\right) + \sum_{l \in I_m^f}\sum_{d \in DI_{m,l}}\left(\sum_{j=1}^{S_l} iw_{i,j}^{m,l}(d)p_j^l(t-d)\right) + b_i^m \tag{5.4}$$

Therefore,

$$\frac{\partial^e n_i^m(t)}{\partial lw_{i,j}^{m,l}(d)} = a_j^l(t-d)\,,\ \frac{\partial^e n_i^m(t)}{\partial iw_{i,j}^{m,l}(d)} = p_j^l(t-d)\,,\ \frac{\partial^e n_i^m(t)}{\partial b_i^m} = 1 \tag{5.5}$$

We will also define

$$d_i^m(t) = \sum_{u \in U} \sum_{k=1}^{S_u} \frac{\partial F}{\partial a_k^u(t)} \times \frac{\partial^e a_k^u(t)}{\partial n_i^m(t)} \tag{5.6}$$

The terms of the gradient can then be written

$$\frac{\partial F}{\partial l w_{i,j}^{m,l}(d)} = \sum_{t=1}^{Q} d_i^m(t) a_j^l(t-d), \tag{5.7}$$

$$\frac{\partial F}{\partial i w_{i,j}^{m,l}(d)} = \sum_{t=1}^{Q} d_i^m(t) p_j^l(t-d), \tag{5.8}$$

$$\frac{\partial F}{\partial b_i^m} = \sum_{t=1}^{Q} d_i^m(t). \tag{5.9}$$

If we use the sensitivity term $s_{k,i}^{u,m}(t)$ defined in Eq. (4.9) and Eq. (4.10), the elements $d_i^m(t)$ can be written

$$d_i^m(t) = \sum_{u \in U} \sum_{k=1}^{S_u} \frac{\partial F}{\partial a_k^u(t)} \times s_{k,i}^{u,m}(t). \tag{5.10}$$

In matrix form this becomes

$$\mathbf{d}^m(t) = \sum_{u \in U} [\mathbf{S}^{u,m}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^u(t)} \tag{5.11}$$

where

$$\frac{\partial F}{\partial \mathbf{a}^u(t)} = \begin{bmatrix} \frac{\partial F}{\partial a_1^u(t)} \\ \frac{\partial F}{\partial a_2^u(t)} \\ \cdots \\ \frac{\partial F}{\partial a_{S_u}^u(t)} \end{bmatrix} \tag{5.12}$$

Now the gradient can be written in matrix form.

$$\frac{\partial F}{\partial \mathbf{LW}^{m,l}(d)} = \sum_{t=1}^{Q} \mathbf{d}^{m}(t) \times [\mathbf{a}^{l}(t-d)]^{T}, \tag{5.13}$$

$$\frac{\partial F}{\partial \mathbf{IW}^{m,l}(d)} = \sum_{t=1}^{Q} \mathbf{d}^{m}(t) \times [\mathbf{p}^{l}(t-d)]^{T}, \tag{5.14}$$

$$\frac{\partial F}{\partial \mathbf{b}^{m}} = \sum_{t=1}^{Q} \mathbf{d}^{m}(t). \tag{5.15}$$

Eq. (5.13) through Eq. (5.15) make up the generalization of Eq. (3.52) for the LDDN network. It remains to compute the explicit derivatives of the sensitivity matrix $\mathbf{S}^{u,m}(t)$. This procedure was described in section 4.4.

5.2. Eq. (3.54).

The next step of the development of the BTT algorithm is the generalization of Eq. (3.54). Again, we use the chain rule:

$$\frac{\partial F}{\partial \mathbf{a}^{u}(t)} = \frac{\partial^{e} F}{\partial \mathbf{a}^{u}(t)} + \sum_{u' \in U} \sum_{x \in X} \sum_{d \in DL_{x,u}} \left[\frac{\partial^{e} \mathbf{a}^{u'}(t+d)}{\partial \mathbf{n}^{x}(t+d)^{T}} \times \frac{\partial^{e} \mathbf{n}^{x}(t+d)}{\partial \mathbf{a}^{u}(t)^{T}} \right]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{u'}(t+d)} \tag{5.16}$$

(Many of the terms in these summations will be zero. We will provide a more efficient representation later in this section.) In Eq. (3.54) we only had one delay in the system. Now we need to account for each network output, how that network output is connected back through a network input, and also for the number of times each network output is delayed before it is applied to a network input. That is the reason for the three summations in Eq. (5.16). This equation must be updated backward in time, as t is varied from Q to 1. The terms

$$\frac{\partial F}{\partial \mathbf{a}^{u'}(t)} \tag{5.17}$$

are generally set to zero for $t > Q$.

If we consider one element of the matrix in the brackets on the right side of Eq. (5.16), we can write

$$\left[\frac{\partial^e \mathbf{a}^{u'}(t+d)}{\partial \mathbf{n}^x(t+d)^T} \times \frac{\partial^e \mathbf{n}^x(t+d)}{\partial \mathbf{a}^u(t)^T}\right]_{i,j} = \sum_{k=1}^{S_x} \frac{\partial^e a_i^{u'}(t+d)}{\partial n_k^x(t+d)} \times \frac{\partial^e n_k^x(t+d)}{\partial a_j^u(t)} \tag{5.18}$$

The first term on the right hand side of this equation is just our previously defined sensitivity

$$\frac{\partial^e a_i^{u'}(t+d)}{\partial n_k^x(t+d)} = s_{i,k}^{u',x}(t+d), \tag{5.19}$$

which can be computed from Eq. (4.24). To find the second term on the right hand side of Eq. (5.18), we can write

$$\begin{aligned} n_k^x(t+d) = {} & \sum_{l \in L_x^f} \sum_{d' \in DL_{x,l}} \left(\sum_{i=1}^{S_l} lw_{k,i}^{x,l}(d') a_i^l(t+d-d') \right) \\ & + \sum_{l \in L_x^f} \sum_{d' \in DI_{x,l}} \left(\sum_{i=1}^{S_l} iw_{k,i}^{x,l}(d') p_i^l(t+d-d') \right) + b_k^x \end{aligned} \tag{5.20}$$

We can now write

$$\frac{\partial^e n_k^x(t+d)}{\partial a_j^u(t)} = lw_{k,j}^{x,u}(d) \tag{5.21}$$

Therefore, Eq. (5.18) can be written

$$\left[\frac{\partial^e \mathbf{a}^{u'}(t+d)}{\partial \mathbf{n}^x(t+d)^T} \times \frac{\partial^e \mathbf{n}^x(t+d)}{\partial \mathbf{a}^u(t)^T}\right]_{i,j} = \sum_{k=1}^{S_x} s_{i,k}^{u',x}(t+d) \times lw_{k,j}^{x,u}(d) \tag{5.22}$$

or in matrix form

$$\frac{\partial^e \mathbf{a}^{u'}(t+d)}{\partial \mathbf{n}^x(t+d)^T} \times \frac{\partial^e \mathbf{n}^x(t+d)}{\partial \mathbf{a}^u(t)^T} = \mathbf{S}^{u',x}(t+d) \times \mathbf{LW}^{x,u}(d). \tag{5.23}$$

This allows us to write Eq. (5.16) as

$$\frac{\partial F}{\partial \mathbf{a}^u(t)} = \frac{\partial^e F}{\partial \mathbf{a}^u(t)} + \sum_{u' \in U} \sum_{x \in X} \sum_{d \in DL_{x,u}} [\mathbf{S}^{u',x}(t+d) \times \mathbf{LW}^{x,u}(d)]^T \times \frac{\partial F}{\partial \mathbf{a}^{u'}(t+d)} \tag{5.24}$$

Many of the terms in the summation on the right hand side of Eq. (5.24) will be zero and will not have to be computed. In order to provide a more efficient implementation of Eq. (5.24), we define the following sets (in addition to those defined in Eq. (4.14) - Eq. (4.16)):

$$E_{LW}^{X}(u) = \{x \in X \ni \exists(\mathbf{LW}^{x,u} \neq 0)\} \tag{5.25}$$

$$E_{S}^{U}(x) = \{u \in U \ni \exists(\mathbf{S}^{u,x} \neq 0)\} \tag{5.26}$$

We can now rearrange the order of the summation in Eq. (5.24):

$$\frac{\partial F}{\partial \mathbf{a}^{u}(t)} = \frac{\partial^{e} F}{\partial \mathbf{a}^{u}(t)} + \sum_{x \in X} \sum_{d \in DL_{x,u}} \sum_{u' \in U} \mathbf{LW}^{x,u}(d)^{T} \times \mathbf{S}^{u',x}(t+d)^{T} \times \frac{\partial F}{\partial \mathbf{a}^{u'}(t+d)} \tag{5.27}$$

and sum only over the existing terms:

$$\frac{\partial F}{\partial \mathbf{a}^{u}(t)} = \frac{\partial^{e} F}{\partial \mathbf{a}^{u}(t)} + \sum_{x \in E_{LW}^{X}(u)} \sum_{d \in DL_{x,u}} \mathbf{LW}^{x,u}(d)^{T} \sum_{u' \in E_{S}^{U}(x)} \mathbf{S}^{u',x}(t+d)^{T} \times \frac{\partial F}{\partial \mathbf{a}^{u'}(t+d)} \tag{5.28}$$

5.3. BTT Gradient Algorithm Summary

The total BTT algorithm is summarized in Figure 5.1

Backpropagation-Through-Time Gradient

```
Initialize:
    $\frac{\partial F}{\partial \mathbf{a}^u(t)} = \mathbf{0}, t > Q$, for all $u \in U$,
For $t = Q$ to 1,
    $U' = \varnothing$, and $E_S(u) = \varnothing$ for all $u \in U$.
    For $m$ decremented through the BP order
        For all $u \in U'$
            $\mathbf{S}^{u,m}(t) = \left[ \sum_{l \in E_S(u) \cap L_m^b} \mathbf{S}^{u,l}(t)\mathbf{LW}^{l,m}(0) \right] \dot{\mathbf{F}}^m(\mathbf{n}^m(t))$
            add $m$ to the set $E_S(u)$
        EndFor $u$
        If $m \in U$
            $\mathbf{S}^{m,m}(t) = \dot{\mathbf{F}}^m(\mathbf{n}^m(t))$
            add $m$ to the sets $U'$ and $E_S(m)$
        EndIf $m$
    EndFor $m$
    For $u \in U$ decremented through the BP order
        $\frac{\partial F}{\partial \mathbf{a}^u(t)} = \frac{\partial^e F}{\partial \mathbf{a}^u(t)} + \sum_{x \in E_{LW}^X(u)} \sum_{d \in DL_{x,u}} \mathbf{LW}^{x,u}(d)^T \sum_{u' \in E_S^U(x)} \mathbf{S}^{u',x}(t+d)^T \times \frac{\partial F}{\partial \mathbf{a}^{u'}(t+d)}$
    EndFor $u$
    For all layers $m$
        $\mathbf{d}^m(t) = \sum_{u \in E_S^U(m)} [\mathbf{S}^{u,m}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^u(t)}$
    EndFor $m$
EndFor $t$
Compute gradients
    $\frac{\partial F}{\partial \mathbf{LW}^{m,l}(d)} = \sum_{t=1}^{Q} \mathbf{d}^m(t) \times [\mathbf{a}^l(t-d)]^T$
    $\frac{\partial F}{\partial \mathbf{IW}^{m,l}(d)} = \sum_{t=1}^{Q} \mathbf{d}^m(t) \times [\mathbf{p}^l(t-d)]^T$
    $\frac{\partial F}{\partial \mathbf{b}^m} = \sum_{t=1}^{Q} \mathbf{d}^m(t)$
```

Figure 5.1: *Pseudo Code for the Backpropagation-Through-Time gradient algorithm*

5.4. BTT Jacobian Algorithm.

In section 4.7 we presented the FP calculation of the Jacobian (defined in Eq. (4.36)). In this section we present the BTT calculation of the Jacobian. In the FP gradient calculation, the elements of the Jacobian (as shown in Eq. (4.38)) were also computed. For this reason, the FP Jacobian calculation was a simplified version of the FP gradient calculation. This is not true for the BTT algorithm.

As shown in Eq. (5.17), the key term computed in the BTT gradient algorithm is $\partial F/\partial \mathbf{a}^{u'}(t)$. In order to compute the Jacobian, we want to replace this term with $\partial v_k(t)/\partial \mathbf{a}^{u'}(t')$. Eq. (5.16) can then be modified as follows:

$$\frac{\partial v_k(t)}{\partial \mathbf{a}^{u}(t')} = \frac{\partial^e v_k(t)}{\partial \mathbf{a}^{u}(t')} + \sum_{u' \in U} \sum_{x \in X} \sum_{d \in DL_{x,u}} \left[\frac{\partial^e \mathbf{a}^{u'}(t'+d)}{\partial \mathbf{n}^{x}(t'+d)^T} \times \frac{\partial^e \mathbf{n}^{x}(t'+d)}{\partial \mathbf{a}^{u}(t')^T}\right]^T \times \frac{\partial v_k(t)}{\partial \mathbf{a}^{u'}(t'+d)} \tag{5.29}$$

where this equation must be updated backward in time, as t' is varied from t to 1. The terms

$$\frac{\partial v_k(t)}{\partial \mathbf{a}^{u'}(t')} \tag{5.30}$$

are set to zero for $t' > t$. The explicit derivative for each output layer will be

$$\frac{\partial^e v_k(t)}{\partial a_k^{u}(t')} = \frac{\partial^e(t_k(t) - a_k^{u^*}(t))}{\partial a_k^{u}(t')} = -\frac{\partial^e a_k^{u^*}(t)}{\partial a_k^{u}(t')} = \begin{cases} -1 & if\ t = t' \\ 0 & if\ t \neq t' \end{cases} \tag{5.31}$$

where $a_k^{u^*}(t)$ are the output layers being used as targets.

Applying a similar development to the one shown in Eq. (5.18) through Eq. (5.23), and the simplification mentioned before Eq. (5.28), we have that

$$\frac{\partial a_k^{u^*}(t)}{\partial \mathbf{a}^{u}(t')} = \frac{\partial^e a_k^{u^*}(t)}{\partial \mathbf{a}^{u}(t')} + \sum_{x \in E_{LW}^{X}(u)} \sum_{d \in DL_{x,u}} \mathbf{LW}^{x,u}(d)^T \sum_{u' \in E_S^{U}(x)} \mathbf{S}^{u',x}(t'+d)^T \times \frac{\partial a_k^{u^*}(t)}{\partial \mathbf{a}^{u'}(t'+d)} \tag{5.32}$$

where the explicit derivatives are based on Eq. (5.31). Eq. (5.32) must be solved backwards in time as t' is varied from t to 1.

We next want to use the terms $\partial a_k^{u^*}(t)/\partial \mathbf{a}^u(t')$ to compute the elements of the Jacobian, $\partial a_k^{u^*}(t)/\partial w_j$. This can be done by replacing the cost function F with the outputs being used as target $a_k^{u^*}(t)$ in Eq. (5.1) through Eq. (5.3):

$$\frac{\partial a_k^{u^*}(t)}{\partial lw_{i,j}^{m,l}(d)} = \sum_{t'=1}^{t} \left\{ \sum_{u \in U} \sum_{k=1}^{S_u} \frac{\partial a_k^{u^*}(t)}{\partial a_k^u(t')} \times \frac{\partial^e a_k^u(t')}{\partial n_i^m(t')} \right\} \frac{\partial^e n_i^m(t')}{\partial lw_{i,j}^{m,l}(d)} \tag{5.33}$$

$$\frac{\partial a_k^{u^*}(t)}{\partial iw_{i,j}^{m,l}(d)} = \sum_{t'=1}^{t} \left\{ \sum_{u \in U} \sum_{k=1}^{S_u} \frac{\partial a_k^{u^*}(t)}{\partial a_k^u(t')} \times \frac{\partial^e a_k^u(t')}{\partial n_i^m(t')} \right\} \frac{\partial^e n_i^m(t')}{\partial iw_{i,j}^{m,l}(d)} \tag{5.34}$$

$$\frac{\partial a_k^{u^*}(t)}{\partial b_i^m} = \sum_{t'=1}^{t} \left\{ \sum_{u \in U} \sum_{k=1}^{S_u} \frac{\partial a_k^{u^*}(t)}{\partial a_k^u(t')} \times \frac{\partial^e a_k^u(t')}{\partial n_i^m(t')} \right\} \frac{\partial^e n_i^m(t')}{\partial b_i^m} \tag{5.35}$$

Applying a similar development to the one shown in Eq. (5.4) through Eq. (5.15) we have that

$$\mathbf{d}_k^m(t, t') = \sum_{u \in U} [\mathbf{S}^{u,m}(t')]^T \times \frac{\partial a_k^{u^*}(t)}{\partial \mathbf{a}^u(t')} \tag{5.36}$$

and

$$\frac{\partial a_k^{u^*}(t)}{\partial \mathbf{LW}^{m,l}(d)} = \sum_{t'=1}^{t} \mathbf{d}^m(t, t') \times [\mathbf{a}^l(t'-d)]^T, \tag{5.37}$$

$$\frac{\partial a_k^{u^*}(t)}{\partial \mathbf{IW}^{m,l}(d)} = \sum_{t'=1}^{t} \mathbf{d}^m(t, t') \times [\mathbf{p}^l(t'-d)]^T, \tag{5.38}$$

$$\frac{\partial a_k^{u^*}(t)}{\partial \mathbf{b}^m} = \sum_{t'=1}^{t} \mathbf{d}^m(t, t'). \tag{5.39}$$

The combination of Eq. (5.32) and Eq. (5.36) through Eq. (5.39) make up the BTT Jacobian calculation. Unlike the FP algorithm, where the Jacobian calculation is simpler than the gradient calculation, the BTT Jacobian calculation is more complex than the BTT gradient. This is because the FP gradient calculation computes the elements of the Jacobian as an intermediate step, whereas the BTT gradient calculation does not. For the BTT Jacobian, at each point in time we must solve Eq. (5.32) backwards to the first time point.

5.5. BTT Jacobian Algorithm Summary

The total BTT algorithm is summarized in Figure 5.2

```
Backpropagation-Through-Time Jacobian

Initialize:
    $\frac{\partial a_k^{u^*}(t)}{\partial \mathbf{a}^u(t')} = \mathbf{0}, \begin{cases} \forall t > Q \\ \forall t \neq t' \end{cases}; \quad \frac{\partial a_k^{u^*}(t)}{\partial \mathbf{a}^u(t')} = -\mathbf{1}, \{\forall t \le Q \text{ and } t = t', \text{ for all } u \in U,$
For $t = Q$ to 1,
    $U' = \varnothing$, and $E_S(u) = \varnothing$ for all $u \in U$.
    For $m$ decremented through the BP order
        For all $u \in U'$
            $\mathbf{S}^{u,m}(t) = \left\{ \sum_{l \in E_S(u) \cap L_m^b} \mathbf{S}^{u,l}(t)\mathbf{LW}^{l,m}(0) \right\} \dot{\mathbf{F}}^m(\mathbf{n}^m(t))$
            add $m$ to the set $E_S(u)$
        EndFor $u$
        If $m \in U$
            $\mathbf{S}^{m,m}(t) = \dot{\mathbf{F}}^m(\mathbf{n}^m(t))$
            add $m$ to the sets $U'$ and $E_S(m)$
        EndIf $m$
    EndFor $m$
EndFor $t$
For $t = Q$ to 1,
    For all the elements of $a_k^{u^*}(t)$,
        For $u \in U$ decremented through the BP order
            For $t' = t$ to 1,
                $\frac{\partial a_k^{u^*}(t)}{\partial \mathbf{a}^u(t')} = \frac{\partial^e a_k^{u^*}(t)}{\partial \mathbf{a}^u(t')} + \sum_{x \in E_{LW}^X(u)} \sum_{d \in DL_{x,u}} \mathbf{LW}^{x,u}(d)^T \sum_{u' \in E_S^U(x)} \mathbf{S}^{u',x}(t'+d)^T \times \frac{\partial a_k^{u^*}(t)}{\partial \mathbf{a}^{u'}(t'+d)}$
            EndFor $t'$
        EndFor $u$
        For all layers $m$
            For $t' = t$ to 1,
                $\mathbf{d}^m(t,t') = \sum_{u \in E_S^U(m)} [\mathbf{S}^{u,m}(t')]^T \times \frac{\partial a_k^{u^*}(t)}{\partial \mathbf{a}^u(t')}$
            EndFor $t'$
        EndFor $m$
        Compute Jacobian terms
            $\frac{\partial a_k^{u^*}(t)}{\partial \mathbf{LW}^{m,l}(d)} = \sum_{t'=1}^{t} \mathbf{d}^m(t,t') \times [\mathbf{a}^l(t'-d)]^T$
            $\frac{\partial a_k^{u^*}(t)}{\partial \mathbf{IW}^{m,l}(d)} = \sum_{t'=1}^{t} \mathbf{d}^m(t,t') \times [\mathbf{p}^l(t'-d)]^T$
            $\frac{\partial a_k^{u^*}(t)}{\partial \mathbf{b}^m} = \sum_{t'=1}^{t} \mathbf{d}^m(t,t')$
    EndFor $a_k^{u^*}(t)$
EndFor $t$
```

Figure 5.2: *Pseudo Code for the Backpropagation-Through-Time Jacobian algorithm*

Chapter 6

ERROR SURFACE ANALYSIS

In this chapter we will suggest a mechanism that can explain, at least in part, the difficulties that occur in training recurrent networks. Based on our analysis of this mechanism, we will also propose modified training procedures that can provide improved convergence. We will demonstrate the operation of these training procedures on two simple recurrent networks.

6.1. Prelude

We begin with an explanation of how we came across a certain characteristic of the error surfaces of recurrent networks. While training a neural-network-based Model Reference Controller [11], we found that the error sometimes increased during training, although a line search was being executed at each iteration. In order to understand the failure of the line search, we plotted the error surface along the search direction. Typical profiles are shown in Figure 6.1. For the system shown, we have 65 weights being trained. The surface we present is along the direction of search (obtained by the BFGS quasi-Newton algorithm) through a 65-dimensional space. It is clear from these profiles that any standard line search, using a combination of interpolation and sectioning, will have great difficulty in locating the minimum along the search direction. There are many local minima contained in very narrow valleys. (Some of the valleys were found to have widths on the order of 10^{-10}.) In addition, the bottom of the valleys are often cusps. We normally assume that the minimum will occur at the point where the derivative is zero. However, for some of these valleys the derivative continues to increase as we approach the minimum. Even if our line search were to locate the minimum, it is not clear that the minimum represents an optimal weight location. In fact, in the remainder of this chapter we will demonstrate that spurious minima are introduced into the error surface due to characteristics in the input sequence.

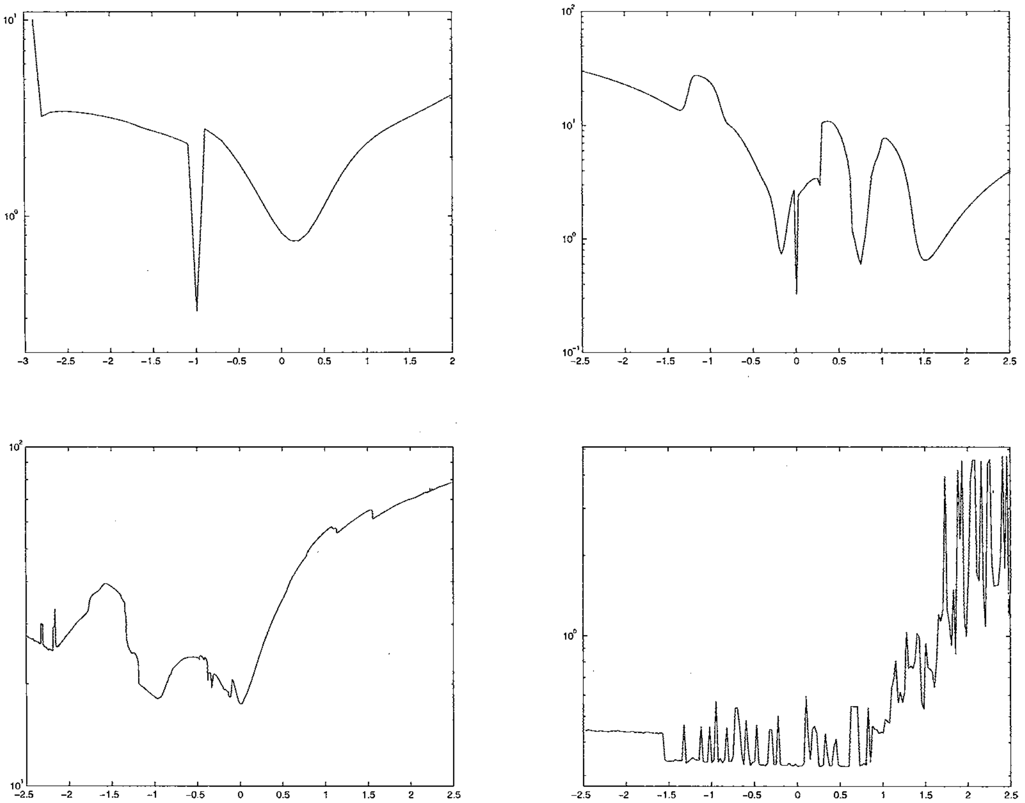

Figure 6.1: *Error profile*

In order to understand how the spurious valleys can appear in the error surface, we analyzed the surfaces for some very simple recurrent networks. The idea was to find the simplest network that would produce the valleys. In the next section we will discuss a first-order linear recurrent network that produces the spurious valleys. We will also show how nonlinear transfer functions can affect the shape of the valleys. This is followed by some modifications we propose to improve the training process, based on our analysis of the creation of the spurious valleys. The fourth section of the chapter tests the proposed modifications on first and second order recurrent networks. In the last section, we give a summary of the results.

6.2. First Order Model

6.2.1. Linear model

Figure 6.2 illustrates the simplest possible recurrent network. As we will see, even this network produces spurious valleys similar to those shown in Figure 6.1.

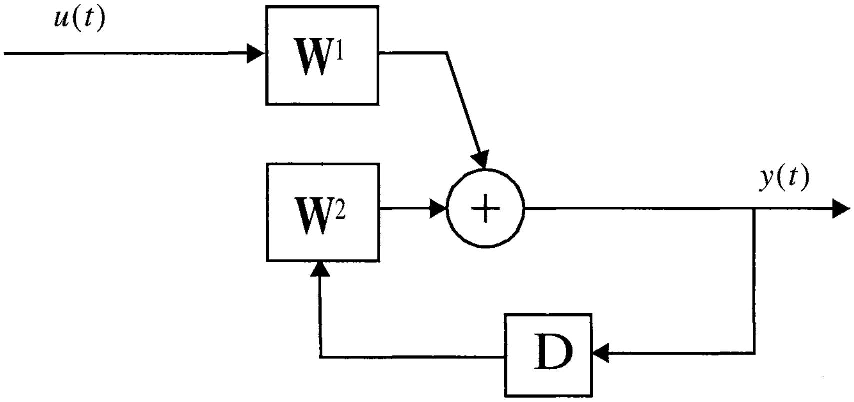

Figure 6.2: *First order linear recurrent network.*

In order to generate an error surface, we first developed training data using the network of Figure 6.2, where the weights were set to $\mathbf{W}^1 = 0.5$ and $\mathbf{W}^2 = 0.5$. We used a random input sequence for $u(t)$, and then used the network to generate a sequence of outputs $y(t)$. Our training objective was then to train another network with the same architecture to fit the training data. The global minimum of the error surface should occur at the values $\mathbf{W}^1 = 0.5$ and $\mathbf{W}^2 = 0.5$.

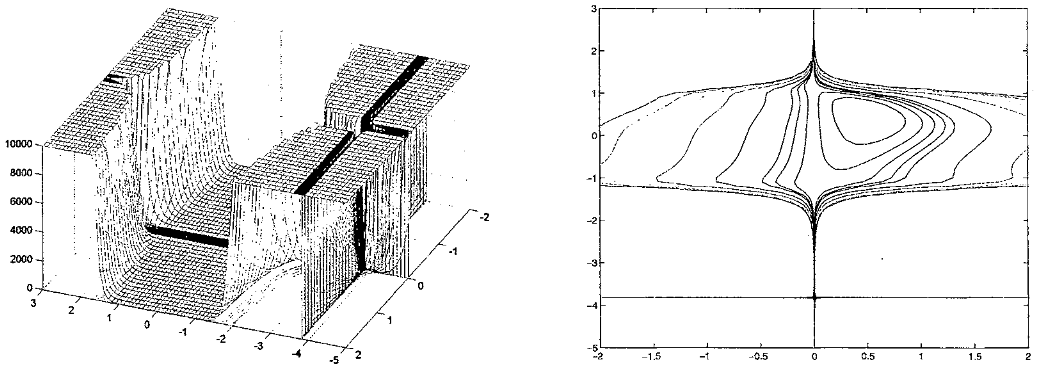

Figure 6.3: *Error surface for first order linear model.*

Figure 6.3 is a typical error surface obtained using the above procedure for one particular input sequence and the initial output $y(0) = 0$. (The plot is cut off at 10,000 to better visualize the surface.) There are several interesting features of the surface. First, the error surface generally increases dramatically as the weight $\mathbf{W}^2$ becomes larger than 1 in magnitude. This is to be expected, since the network is unstable for these weight values. What is unexpected are the two valleys that run through the surface. Even though the network is unstable for $|\mathbf{W}^2| > 1$, for this particular input sequence there are some values for $\mathbf{W}^2$ in this range that produce small network outputs (and therefore relatively small errors). We expect the output to grow without bound under these conditions, but it doesn't always happen.

The two valleys in the error surface occur for two different reasons. One valley occurs at $\mathbf{W}^1 = 0$. If this weight is zero, and the initial condition is zero, the output of the network will remain zero. Therefore, our mean squared error will be constant and equal to the mean square value of the target outputs.

The second valley in the error surface is due to the input sequence that is presented to the network. For a given input $u(t)$, the system output will be:

$$y(t) = \mathbf{W}^1 u(t) + \mathbf{W}^2 y(t-1) \tag{6.1}$$

If we accumulate the responses starting from some initial condition $y(0) = 0$ up to time $k > 0$, we obtain

$$\begin{aligned} y(k) &= \mathbf{W}^1 u(k) + \mathbf{W}^2 y(k-1) \\ &= \mathbf{W}^1 u(k) + \mathbf{W}^2(\mathbf{W}^1 u(k-1) + \mathbf{W}^2 y(k-2)) \\ &\dots \quad \dots \quad \dots \quad \dots \\ &= \mathbf{W}^1\{u(k) + \mathbf{W}^2 u(k-1) + (\mathbf{W}^2)^2 u(k-2) \\ &\quad + \dots + (\mathbf{W}^2)^{k-1} u(1)\} + (\mathbf{W}^2)^k y(0) \end{aligned} \tag{6.2}$$

Here we can see that the response at time k is a polynomial in the parameter $\mathbf{W}^2$. The coefficients of the polynomial involve the input sequence and the initial condition. We obtain the second valley because this polynomial contains a root outside the unit circle. There is some value of $\mathbf{W}^2$ that is larger than 1 in magnitude for which the output $y(k)$ is almost zero.

Of course, having a single output close to zero would not produce a valley in the error surface. However, we discovered that once the polynomial shown in Eq. (6.2) has a root outside the unit circle, that root appears also in the next polynomial at time k+1, and therefore the output will remain small for future times.

Figure 6.4 shows a cross-section of the error surface presented in Figure 6.3 for $\mathbf{W}^1 = 0.5$ using different sequence lengths. The error falls abruptly near -3.8239. That is the root of the polynomial described in Eq. (6.2). The root maintains its location as the sequence increases in length (k increases).

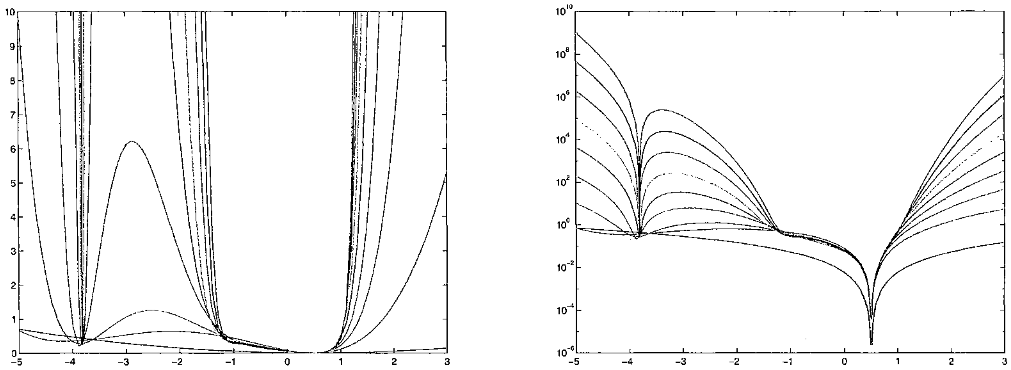

Figure 6.4: *Error cross-section for* $\mathbf{W}^1 = 0.5$.

To summarize, there are two mechanisms that create the spurious valleys. The first mechanism has to do with the initial conditions. If some initial conditions are zero, then there are certain combinations of weights that will produce zero outputs for all time. (This effect is more complex in larger networks, as we will see in a later example.) The second mechanism has to do with the input sequence. There are values for the weights that produce an unstable network, but for which the output remains small for a particular input sequence. If the input sequence is modified, it may produce a valley in a different location.

In section 6.3 we will propose some modified training procedures that can mitigate the effects of the spurious valleys. Before introducing that topic, let's investigate the effect of nonlinear transfer functions on the error surface.

6.2.2. Nonlinear network

Figure 6.5 presents a nonlinear modification of the linear network presented in the previous section. Here we include a sigmoid nonlinearity at the output.

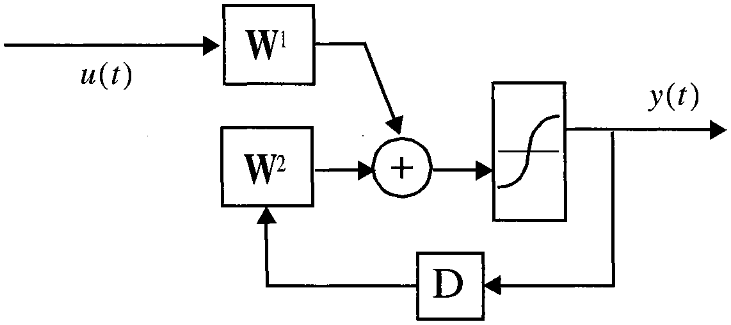

Figure 6.5: *First order nonlinear model.*

Figure 6.6 presents the error surface for the nonlinear network using the same input sequence used in the previous section. Due to the nonlinearity, the output is bounded for large weight values. So the error does not grow without bound, as in the linear network. We notice that the valley is still present, however it is bent. This curving valley is still able to trap the training algorithm and even to move the weights away from the true minimum.

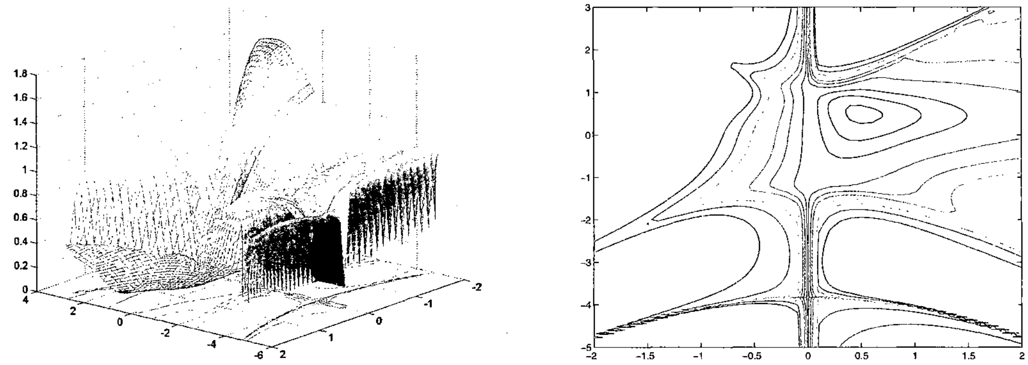

Figure 6.6: *Error surface for first order nonlinear network.*

6.3. Modifications to the Training Procedure

From the previous section we see that difficulties in training recurrent neural networks could be due to the presence of spurious valleys. The shape of the valleys could be complex for large nonlinear neural networks. If a gradient search algorithm falls inside a valley we may converge to a region where the network is unstable or where the weights are unreasonably large. The location of those valleys depends on the input sequence and on the initial conditions. In this section we will propose three modifications to standard training procedures that can mitigate the effects of the valleys.

6.3.1. Proposed solutions

In this section we will propose three variations to the standard training algorithms for recurrent networks. These variations include regularization, switching training sequences, and randomly setting initial conditions.

If we compare the linear and nonlinear cases from section 6.2, we notice that the linear case has a natural way of allowing convergence to the optimal weights, because larger weights generate large outputs.

The farther we move from the stable region, the larger the gradient will become. A gradient descent algorithm would generally move the weights toward the stable region. This effect does not occur in the nonlinear networks. However, we can obtain a similar effect if we combine regularization [38] with our mean square error performance function. In other words we can use the performance function

$$J(\mathbf{W}) = SSE + \alpha SSW, \tag{6.3}$$

where SSE is the sum squared errors and SSW is the sum squared weights. This performance function would help to force the weights back into the stable region, because it would overwhelm the spurious valleys for large values of the weights. We can decrease the regularization factor α during training to ensure that we don't bias the final trained weights.

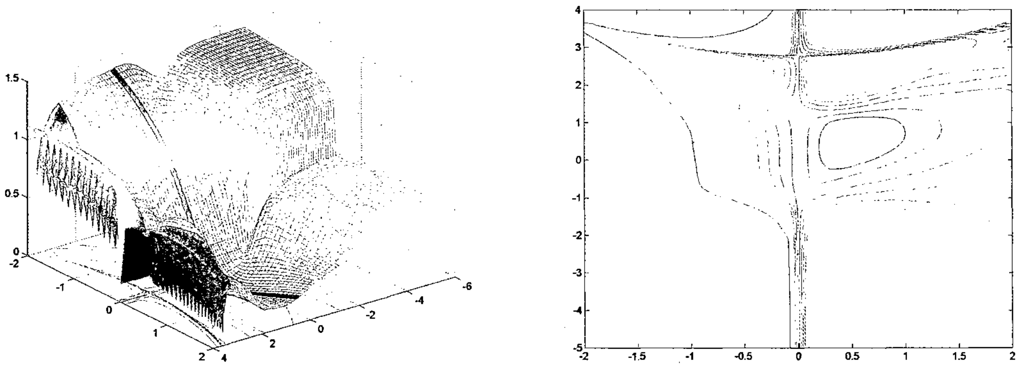

Figure 6.7: *Error surface for first order nonlinear model for different input sequence.*

Another technique for improved training involves using more than one training sequence. Figure 6.7 presents the error surface for the nonlinear model of Figure 6.5, using a different sequence. The valley that appeared in Figure 6.6 has moved to the positive region of $\mathbf{W}^2$. For any two random input sequences, the valleys will appear in different locations.

This suggests another technique for improved training. We could use multiple input sequences. Because valleys are sequence dependent, we can use one sequence for a given number of epochs and then alternate to a new sequence. If we become trapped in a spurious valley, that valley will disappear when the new sequence is presented.

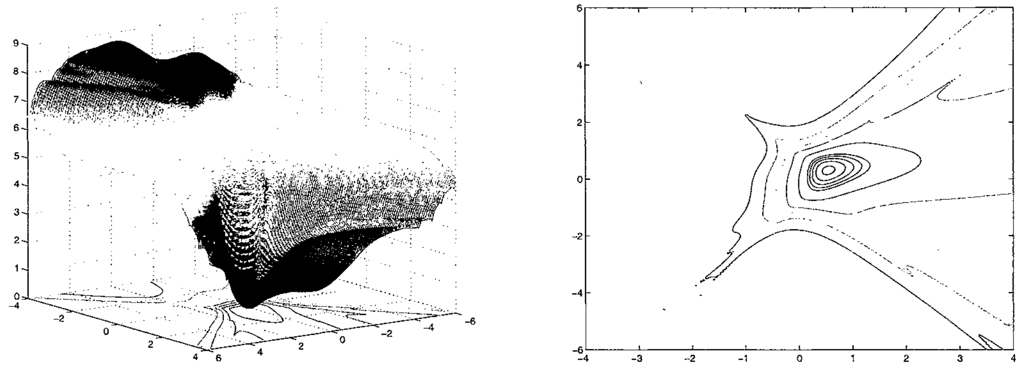

Figure 6.8: *Error surface using sequence averaging.*

Another implementation of multiple sequences could be sequence averaging. We could compute the gradients for multiple sequences and then move in the direction of the average. Figure 6.8 presents an error surface for five sequences. This figure demonstrates how the spurious valleys are reduced in amplitude.

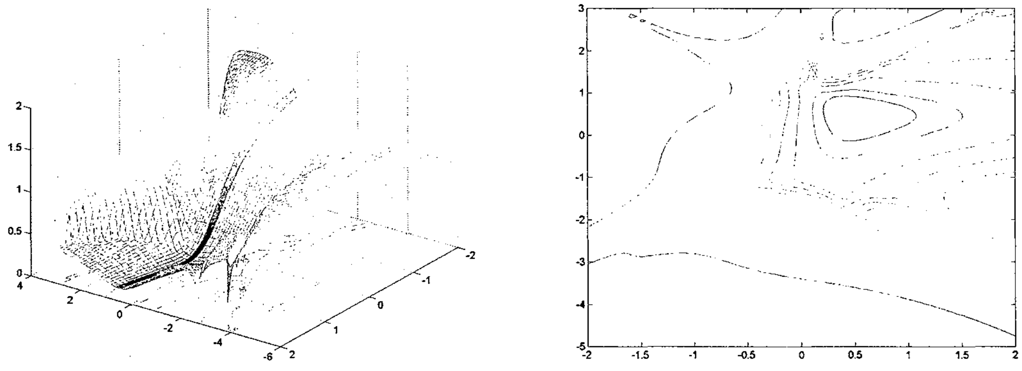

Figure 6.9: *Error surface using* $y(0) = 0.1$.

Another method to move the valleys is to use random initial conditions. Figure 6.9 shows how the error surface is changed when we set the initial condition to $y(0) = 0.1$. The valley at $\mathbf{W}^1 = 0$, which we discussed earlier, is missing. In later experiments with larger networks, we found that the valleys do not always disappear when nonzero initial conditions are used. They are often only moved to new locations. A better approach would be to use different small random initial conditions at different stages of training. We could switch the initial conditions in combination with the switching of sequences.

In all, we have four proposed training modifications. For ease of reference, we will label them as follows: switching sequences (SS), averaging sequences (AS), regularization (REG), nonzero initial conditions (IC).

6.4. Test Results for Gradient Algorithm

In this section we will test the training modifications that were proposed in the previous section using the gradient algorithm described in section 5.3. For these tests we will train the nonlinear network shown in section 6.2 (and a more complex, second-order network) using the standard gradient descent algorithm with a golden section line search. We will not worry about using the most sophisticated training algorithm. Rather, the objective will be to verify the ability of the new procedures to improve training performance. We will define the results obtained with the gradient descent algorithm alone as our baseline. Other tests will be performed for each one of the proposed modifications. For the REG test, we divided α by 1.2 at each epoch. For the IC method we set all layer initial conditions to 0.2. One test was performed using all three methods. We called this training procedure the "Multiple" method. For all tests, the gradient is computed using the dynamic backpropagation method described in section 5.3.

6.4.1. First order nonlinear system

For the first order nonlinear system we generated training data using $\mathbf{W}^1 = 0.5$ and $\mathbf{W}^2 = 0.5$. The training was done using 25000 different sequences of 15 samples each and random initial conditions. The random initial weights were generated in three different levels: 1, 5 and 20 standard deviations from the true solution.

Table 6.1 summarizes the results of the first tests on the first order network. It shows the percentage of tests in which the weights converged close to the optimal weights. Each method provides some improvement on the baseline method. However, the multiple method is the only one that guarantees accurate convergence.

Table 6.1: *Percentage of final weights within 0.001 of the optimal weights for the first order nonlinear network.*

Method	STD of the initial weights		
	1	5	20
Baseline	92.1	61.2	37.9
REG	99.6	99.7	99.9
SS	96.5	64.7	45.7
AS	94.3	58.1	42.7
IC	95.6	71.1	45.0
Multiple	100.0	100.0	100.0

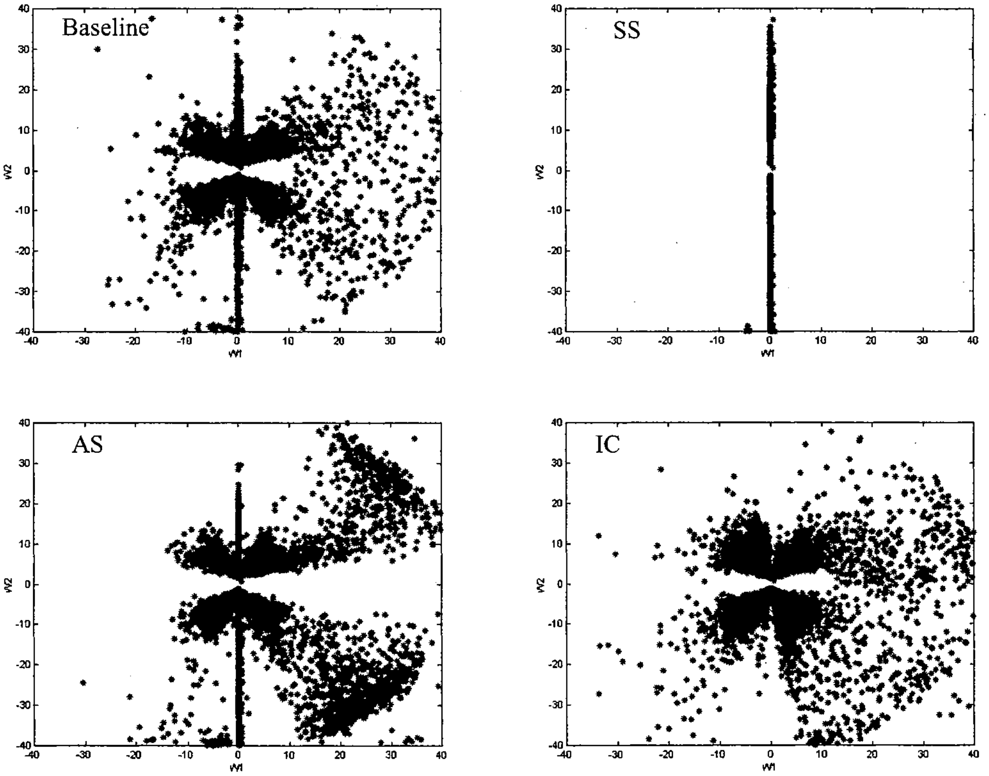

Figure 6.10: *Relative final position of* $\mathbf{W}^1$ *vs.* $\mathbf{W}^2$ *for 5 std.*

Figure 6.10 shows the relative final position of $\mathbf{W}^1$ vs. $\mathbf{W}^2$ for Baseline, SS, AS and IC. For the first three methods many tests finished along $\mathbf{W}^1 = 0$. That condition was removed when we set the initial conditions to 0.2. When we switch the sequences we avoided many cases where training may be trapped in the spurious valleys. The averaging of sequences did not improve our training results, resulting in worse results than the baseline method for 5 std.

6.4.2. Two layer neural network

Figure 6.11 has a neural network with two layers, where each layer is fed back to the previous layers. This system will allow us to test the previous training procedure modifications on a more complex system. For these tests, we generated training data using the following weights:

$$\begin{array}{ll} \mathbf{W}^2 = -0.5 & \mathbf{W}^3 = 0.25 \\ \mathbf{W}^1 = 0.5 & \mathbf{W}^4 = -0.3 \end{array} \tag{6.4}$$

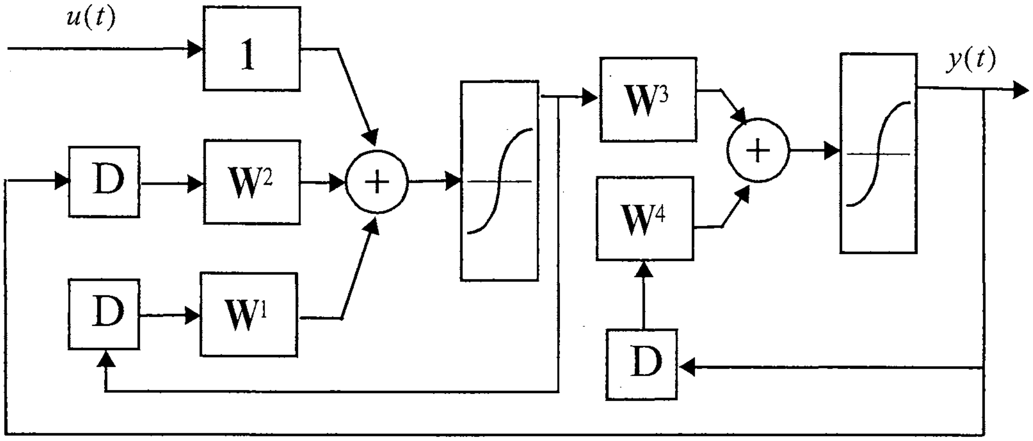

Figure 6.11: *Two-layer nonlinear model.*

Table 6.2 shows the percentage of weights close to the final weights after the training process. For this neural network architecture, regularization resulted in a success rate of over ninety percent. However, it is again the multiple method that guarantees the best convergence.

Table 6.2: *Percentage of final weights within 0.5 of the optimal weights for the two-layer nonlinear network.*

Method	STD of the initial weights		
	1	5	20
Baseline	82.2	12.8	0.3
REG	93.0	95.0	97.0
SS	95.8	38.6	2.5
IC	54.6	7.2	0
Multiple	100.0	99.0	100.0

Figure 6.12 presents the final weight positions in the $\mathbf{W}^3$ vs. $\mathbf{W}^4$ plane for the Baseline, SS and IC training methods. For the Baseline training method, we notice the presence of three axes or valleys where the training converged. From the middle figure we can see that the SS method can eliminate the diagonal final condition. However, the axis along $\mathbf{W}^3 = 0$ remains. When we set the initial layer conditions to 0.2, we can see from the last figure that two new axes appear. This demonstrates that setting the initial conditions to nonzero values does not necessarily remove spurious valleys. It may just move them to new locations. This suggests that we should vary the initial conditions whenever we switch the training sequence.

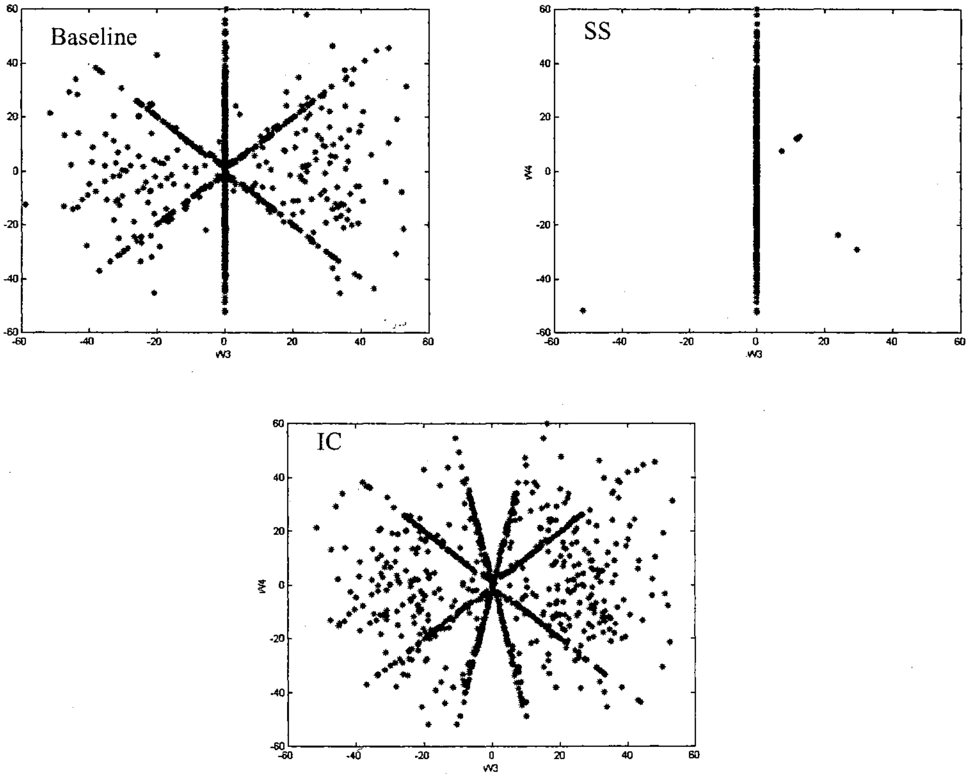

Figure 6.12: *Relative final position of* $\mathbf{W}^3$ *vs.* $\mathbf{W}^4$ *for 20 std.*

Figure 6.13 shows how the final distance to the optimal weights is affected by the switching sequence interval. While training for 10000 epochs, we switched the training sequence every 1, 10, 100, 500 and 1000 epochs. Frequent changes consistently resulted in more accurate final weights. If training continues with the same sequence, we could be caught in a spurious valley, resulting in failed training.

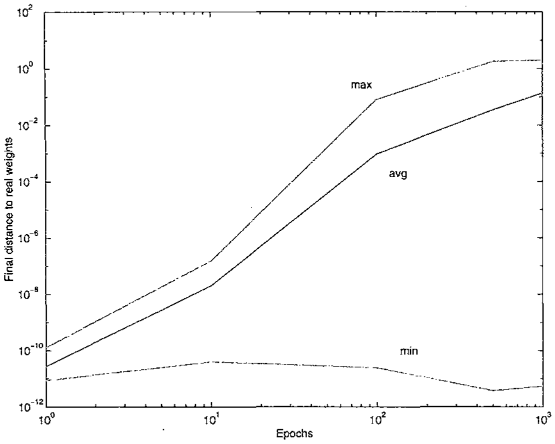

Figure 6.13: *Final distance to optimal weights for different switching sequence intervals.*

Figure 6.14 shows the average performance for three different switching intervals. We obtain substantial improvement when the sequence is switched more frequently. We can conclude that we should not maintain the same sequence for long periods, when training recurrent neural networks.

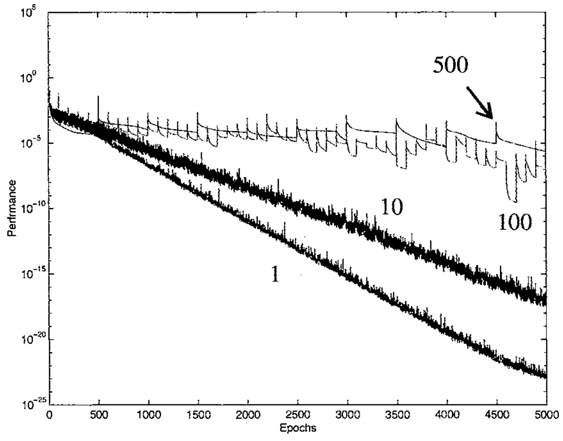

Figure 6.14: *Average performance for different switching sequence interval.*

Another battery of tests was performed to evaluate how to adjust α when regularization is being used. We adjusted α by dividing it by a constant at each epoch. The constants we used were 1.01, 1.2 and 2. Figure 6.15 shows the average performance when α is divided by 1.01 and 1.2. The best results were obtained for 1.2. (The results for 2 were almost identical to the results for 1.2.) From this test we can conclude that α must be decreased in some way to obtain the best training results.

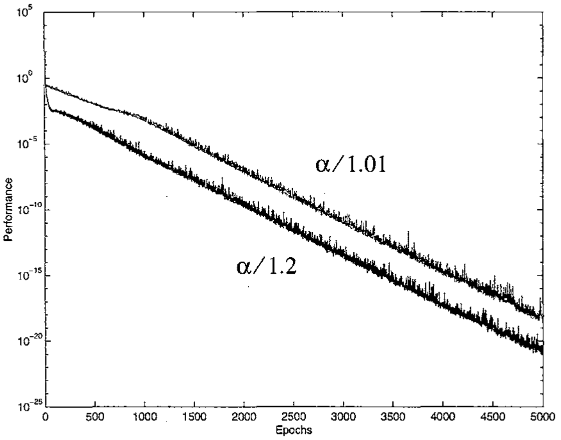

Figure 6.15: *Average performance for* α/1.01 *and* α/1.2.

Figure 6.16 shows the number of flops required to train the two-layer neural network to convergence using the multiple method with different sequence lengths. This figure does not demonstrate any advantage to using long sequences for this network. The algorithm converged for all sequences, but the longer sequences require more computation. One would expect that for more complex networks there might be some advantage to longer sequences.

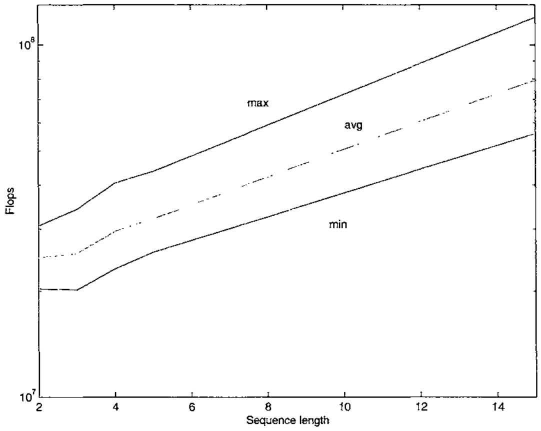

Figure 6.16: *Flops for different sequence length.*

6.5. Test Results for Jacobian Algorithm

In this section we will test the training modifications that were proposed in section 6.3 using the Jacobian algorithm from section 4.8. For these tests we will train the nonlinear network shown in section 6.2 (and a more complex, second-order network) using the Levenberg-Marquardt algorithm. Like the previous section, the objective will be to verify the ability of the new procedures to improve training performance. We will define the results obtained with the gradient descent algorithm alone as our baseline. Other tests will be performed for each one of the proposed modifications based on the Levenberg-Marquardt features. For the REG test, we used the Bayesian regularization procedure proposed by Foresee and Hagan [16], and Chen and Hagan [6]. One test was performed using all three methods. We called this training procedure the "Multiple" method. For all tests, the Jacobian is computed using the dynamic backpropagation method described in section 4.8.

6.5.1. First order nonlinear system

For the first order nonlinear system we generated training data using $\mathbf{W}^1 = 0.5$ and $\mathbf{W}^2 = 0.5$. The training was done using 25000 different sequences of 15 samples each and random initial conditions. The random initial weights were generated in three different levels: 1, 5 and 20 standard deviations from the true solution.

Table 6.3 summarizes the results of the first tests on the first order network. It shows the percentage of tests in which the weights converged close to the optimal weights. Each method provides some improvement on the baseline method. However, the multiple method is the only one that guarantees accurate convergence. If we compare the Jacobian results against the gradient results presented in Table 6.1, we could confirm that the Jacobian-based algorithm outperforms gradient-based algorithms. The criteria for Jacobian algorithms was two orders of magnitude more strict than gradient algorithms. Even under those conditions, the results were better for similar testing conditions.

Table 6.3: *Percentage of final weights within 0.00001 of the optimal weights for the first order nonlinear network.*

Method	STD of the initial weights		
	1	5	20
Baseline	98.9	92.4	69.2
REG	99.7	98.9	96.1
SS	99.8	97.4	84.8
IC	99.4	93.7	72.1
Multiple	100.0	99.9	99.9

6.5.2. Two layer neural network

This section tested the Jacobian-based algorithms using a neural network with two layers, where each layer is fed back to the previous layers, as shown in Figure 6.11. For these tests, we generated training data using the weights presented in Eq. (6.4).

Table 6.4 shows the percentage of weights close to the final weights after the training process. For this neural network architecture, regularization resulted in a success rate of over ninety percent with initial weights up to 5 standard deviations from the true weights (even though using strict performance conditions than for the gradient based algorithms). However, it is again the multiple method that guarantees the best convergence. If we compare the results against the gradient based algorithms in Table 6.2, we notice better results provided by Jacobian-based algorithms.

Table 6.4: *Percentage of final weights within 0.00001 of the optimal weights for the second order nonlinear network.*

Method	STD of the initial weights		
	1	5	20
Baseline	71.7	33.9	2.8
REG	99.7	97.9	71.2
AS	96.0	54.0	9.6
IC	94.0	54.2	6.6
Multiple	100.0	99.0	85.0

6.6. Summary

This chapter suggested that spurious valleys are present in the error surface of dynamic networks. Those narrow valleys often trap gradient based training algorithms. We suggested some procedures to improve convergence: switching sequences (SS), regularization (REG) and nonzero initial conditions (IC).

The best results were obtained when all procedures were used simultaneously. The previous procedures were demonstrated using two simple recurrent networks.

Chapter 7

APPLICATIONS

This chapter will present two types of applications for dynamic neural networks. The first class of applications is Model Reference Control. For this type of controller architectures, the plant is modelled using a feedforward neural network. Later, the plant model network is used in combination with a separate neural network controller. The plant model network is used to backpropagate the controller errors and update the controller weights. Due to the feedback nature of the controller and plant combination, dynamic backpropagation is required to update the controller weights. The second class of applications involve filter design. We will use dynamic neural networks to emulate a nonlinear process.

We will use these two classes of applications to show how the dynamic backpropagation methods are applied. For the controller we will use the FP algorithm. For the filter example we will use the BTT algorithm.

7.1. Model Reference Control.

In this section we illustrate the application of the LDDN and dynamic backpropagation to different control problems. We will describe the Model Reference Controller [35] and apply it to four different systems: a continuous stirred tank reactor, a single-link robot arm, a magnetic levitation system and a simple diesel engine model. These four systems represent a variety of simple applications to which neurocontrollers can be applied. We will then compare and contrast the performances on the four test problems.

There are typically two steps involved when using neural networks for control: system identification and control design. In the system identification stage, you develop a neural network model of the plant that you want to control. This network is trained offline in batch mode, using data collected from the operation of the plant. In the control design stage, you use the neural network plant model to design (or train) the controller. The control design stage, however, changes for different controller implementations. Figure 7.1 shows the architecture of the Model Reference Controller.

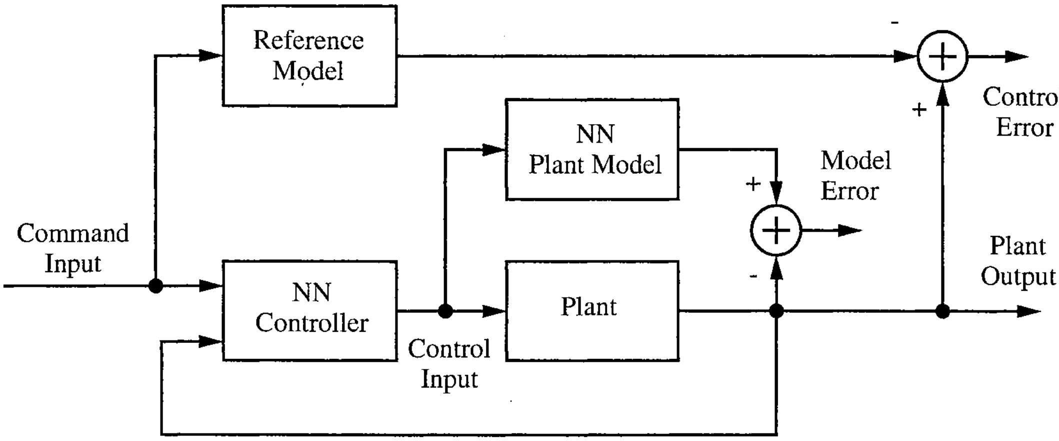

Figure 7.1: *Model Reference Control Architecture*

The model reference architecture [35] requires that a separate neural network controller be trained off-line, in addition to the neural network plant model. The controller training is computationally expensive, since it requires the use of dynamic backpropagation [35, 19]. On the positive side, model reference control applies to a larger class of plant than, for example, does NARMA-L2 control (see Eq. (2.6)). Each network has two layers, and you can select the number of neurons to use in the hidden layers. There are three sets of delayed controller inputs: reference inputs, controller outputs and plant outputs. Table 9.1 shows the final Model Reference controller parameters for each test.

Table 7.1: *Parameters for Model Reference Controller*

	CSTR	Maglev	Robot	Engine
Del. reference	3	2	2	2
Delayed inputs	3	2	2	2
Delay. outputs	3	2	1	1
Hidden layer	8	13	13	15

The following figure shows the details of the neural network plant model and the neural network controller. Each network has two layers, and you can select the number of neurons to use in the hidden layers. The plant model is used only as a backpropagation path for the derivatives needed to adjust the controller weights; the plant model weights are not adjusted. There are three sets of controller inputs: delayed reference inputs, delayed controller outputs and delayed plant outputs. For each of these inputs, you can select the number of delayed values to use. Typically, the number of delays increases with the order of the plant.

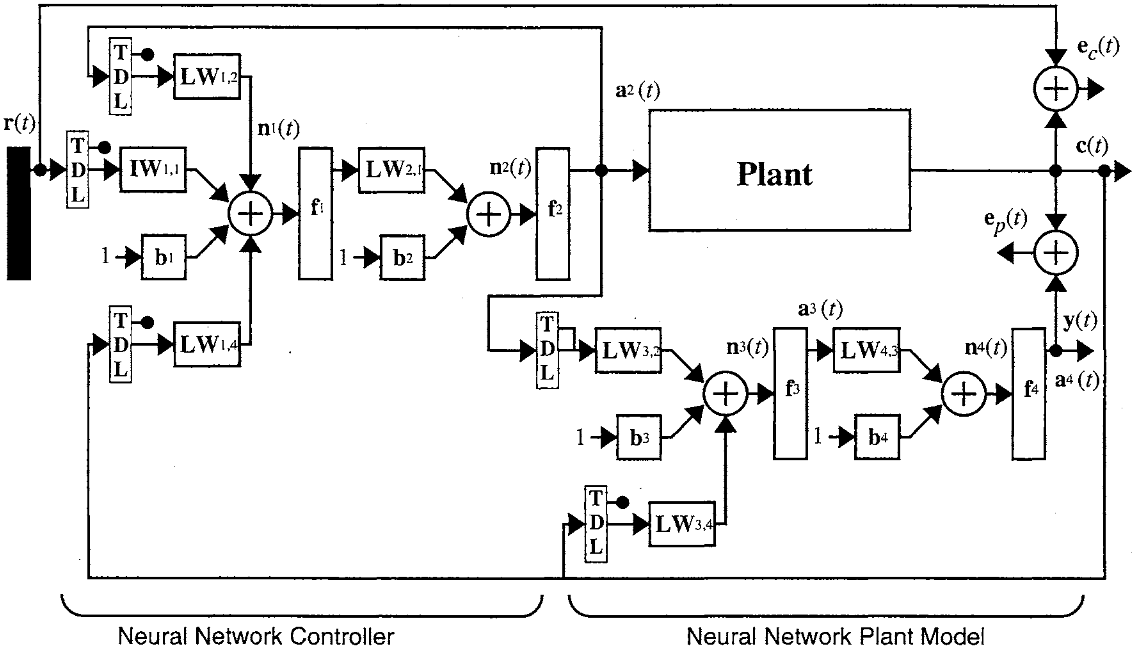

Figure 7.2: *Detailed Model Reference Control Structure*

If we apply the algorithm described in Figure 4.2 to the system shown in Figure 7.2, we first notice that the Backpropagation order is 4-3-2-1, the set of all output layer numbers is U={2,4} and the set of all input layer numbers is X={1,3}.

We now show how these equations can be developed using our general FP algorithm. We start in the last layer in the backpropagation order (layer 4), obtaining:

$$\mathbf{S}^{4,4}(t) = \dot{\mathbf{F}}^{4}(\mathbf{n}^{4}(t))\,;\; U' = 4\;;\; E_S(4) = 4$$

Layer 3 is only connected to layer 4 through a non-delay, so we apply:

$$\mathbf{S}^{4,3}(t) = \mathbf{S}^{4,4}(t)\mathbf{LW}^{4,3}(0)\dot{\mathbf{F}}^{3}(\mathbf{n}^{3}(t))\,;\; E_S(3) = 3\;;\; E_S^X(4) = 3$$

Layer 2 is the last layer of the neural controller and it is connected to another layer through delays. So we apply the equations:

$$\mathbf{S}^{2,2}(t) = \dot{\mathbf{F}}^{2}(\mathbf{n}^{2}(t))\,;\; U' = \{2,4\}\;;\; E_S(2) = 2$$

Layer 1 is only connected to layer 2 through a non-delay, so we apply:

$$\mathbf{S}^{2,1}(t) = \mathbf{S}^{2,2}(t)\mathbf{LW}^{2,1}(0)\dot{\mathbf{F}}^{1}(\mathbf{n}^{1}(t))$$

We can solve now the explicit derivatives for each weight and bias:

$$\frac{\partial^{e}\mathbf{a}^{4}(t)}{\partial(\mathbf{b}^{4})^{T}} = \mathbf{S}^{4,4}(t)\,;\ \frac{\partial^{e}\mathbf{a}^{4}(t)}{\partial vec(\mathbf{LW}^{4,3}(0))^{T}} = [\mathbf{a}^{3}(t)]^{T}\otimes\mathbf{S}^{4,4}(t)\,;\ \frac{\partial^{e}\mathbf{a}^{4}(t)}{\partial(\mathbf{b}^{3})^{T}} = \mathbf{S}^{4,3}(t)$$

$$\frac{\partial^{e}\mathbf{a}^{4}(t)}{\partial vec(\mathbf{LW}^{3,2}(d))^{T}} = \begin{bmatrix}\mathbf{a}^{3}(t)\\ \mathbf{a}^{3}(t-1)\\ \cdots\\ \mathbf{a}^{3}(t-d)\end{bmatrix}^{T}\otimes\mathbf{S}^{4,3}(t)\,;\ \frac{\partial^{e}\mathbf{a}^{4}(t)}{\partial vec(\mathbf{LW}^{3,4}(d))^{T}} = \begin{bmatrix}\mathbf{a}^{4}(t)\\ \mathbf{a}^{4}(t-1)\\ \cdots\\ \mathbf{a}^{4}(t-d)\end{bmatrix}^{T}\otimes\mathbf{S}^{4,3}(t)$$

$$\frac{\partial^{e}\mathbf{a}^{2}(t)}{\partial(\mathbf{b}^{2})^{T}} = \mathbf{S}^{2,2}(t)\,;\ \frac{\partial^{e}\mathbf{a}^{2}(t)}{\partial vec(\mathbf{LW}^{2,1}(0))^{T}} = [\mathbf{a}^{1}(t)]^{T}\otimes\mathbf{S}^{2,2}(t)\,;\ \frac{\partial^{e}\mathbf{a}^{2}(t)}{\partial(\mathbf{b}^{1})^{T}} = \mathbf{S}^{2,1}(t)$$

$$\frac{\partial^{e}\mathbf{a}^{2}(t)}{\partial vec(\mathbf{LW}^{1,2}(d))^{T}} = \begin{bmatrix}\mathbf{a}^{2}(t)\\ \mathbf{a}^{2}(t-1)\\ \cdots\\ \mathbf{a}^{2}(t-d)\end{bmatrix}^{T}\otimes\mathbf{S}^{2,1}(t)\,;\ \frac{\partial^{e}\mathbf{a}^{2}(t)}{\partial vec(\mathbf{LW}^{1,4}(d))^{T}} = \begin{bmatrix}\mathbf{a}^{4}(t)\\ \mathbf{a}^{4}(t-1)\\ \cdots\\ \mathbf{a}^{4}(t-d)\end{bmatrix}^{T}\otimes\mathbf{S}^{2,1}(t)$$

$$\frac{\partial^{e}\mathbf{a}^{2}(t)}{\partial vec(\mathbf{IW}^{1,1}(d))^{T}} = \begin{bmatrix}\mathbf{r}(t)\\ \mathbf{r}(t-1)\\ \cdots\\ \mathbf{r}(t-d)\end{bmatrix}^{T}\otimes\mathbf{S}^{2,1}(t)$$

Now that we have finished with the backpropagation step, we have the explicit derivatives for all of the weights and biases in the system. We are now ready to calculate the dynamic derivatives:

$$\begin{aligned}\frac{\partial\mathbf{a}^{2}(t)}{\partial\mathbf{w}^{T}} &= \frac{\partial^{e}\mathbf{a}^{2}(t)}{\partial\mathbf{w}^{T}} + \sum_{x\in E_{S}^{X}(2)}\mathbf{S}^{u,x}(t)\sum_{u'\in E_{LW}^{U}(x)}\ \sum_{d\in DL_{x,u'}}\mathbf{LW}^{x,u'}(d)\times\frac{\partial\mathbf{a}^{u'}(t-d)}{\partial\mathbf{w}^{T}}\\ &= \frac{\partial^{e}\mathbf{a}^{2}(t)}{\partial\mathbf{w}^{T}} + \mathbf{S}^{2,1}(t)\left\{\sum_{d\in DL_{1,2}}\mathbf{LW}^{1,2}(d)\times\frac{\partial\mathbf{a}^{2}(t-d)}{\partial\mathbf{w}^{T}} + \sum_{d\in DL_{1,4}}\mathbf{LW}^{1,4}(d)\times\frac{\partial\mathbf{a}^{4}(t-d)}{\partial\mathbf{w}^{T}}\right\}\end{aligned}$$

and

$$\frac{\partial \mathbf{a}^4(t)}{\partial \mathbf{w}^T} = \frac{\partial^e \mathbf{a}^4(t)}{\partial \mathbf{w}^T} + \sum_{x \in E_S^X(4)} \mathbf{S}^{u,x}(t) \sum_{u' \in E_{LW}^U(x)} \sum_{d \in DL_{x,u'}} \mathbf{LW}^{x,u'}(d) \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial \mathbf{w}^T}$$

$$= \frac{\partial^e \mathbf{a}^4(t)}{\partial \mathbf{w}^T} + \mathbf{S}^{4,3}(t) \left\{ \sum_{d \in DL_{3,2}} \mathbf{LW}^{3,2}(d) \times \frac{\partial \mathbf{a}^2(t-d)}{\partial \mathbf{w}^T} + \sum_{d \in DL_{3,4}} \mathbf{LW}^{3,4}(d) \times \frac{\partial \mathbf{a}^4(t-d)}{\partial \mathbf{w}^T} \right\}$$

This previous process is repeated for each sample time in the training set.

We have now computed all of the dynamic derivatives of the outputs with respect to the weights. The next step is to compute the derivatives of the performance function with respect to the weights. We must first calculate

$$\frac{\partial^e F}{\partial \mathbf{a}^4(t)} = -2(\mathbf{r}(t) - \mathbf{a}^4(t)),$$

to obtain

$$\frac{\partial F}{\partial \mathbf{w}^T} = \sum_{t=1}^{Q} \sum_{u \in U} \left\{ \left[\frac{\partial^e F}{\partial \mathbf{a}^u(t)} \right]^T \times \frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{w}^T} \right\} = -\sum_{t=1}^{Q} \left\{ [2(\mathbf{r}(t) - \mathbf{a}^4(t))]^T \times \frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{w}^T} \right\}$$

for all the weights and biases in the neural controller. The next section will present the results for the Model Reference Controller for the systems mentioned at the beginning of this section.

7.1.1. Case Studies

7.1.1.1. *Continuous Stirred Tank Reactor (CSTR)*

The first application is a catalytic Continuous Stirred Tank Reactor (CSTR) [5] shown in Figure 7.3.

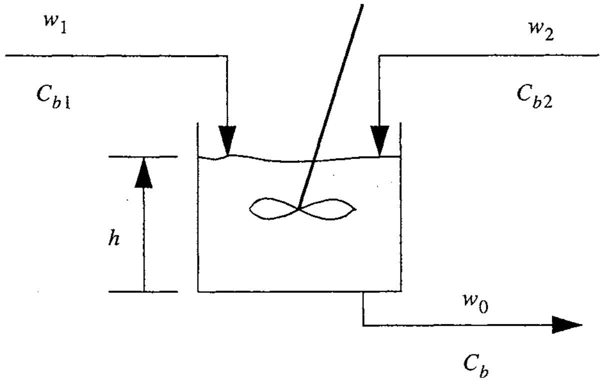

Figure 7.3: *Continuous Stirred Tank Reactor*

The dynamic model of the system is

$$\frac{dh(t)}{dt} = w_1(t) + w_2(t) - 0.2\sqrt{h(t)} \tag{7.1}$$

$$\frac{dC_b(t)}{dt} = (C_{b1}\text{-}C_b(t))\frac{w_1(t)}{h(t)} + (C_{b2}\text{-}C_b(t))\frac{w_2(t)}{h(t)} - \frac{k_1 C_b(t)}{(1+k_2 C_b(t))^2} \tag{7.2}$$

where $h(t)$ is the liquid level, $C_b(t)$ is the product concentration at the output of the process, $w_1(t)$ is the flow rate of the concentrated feed C_{b1}, and $w_2(t)$ is the flow rate of the diluted feed C_{b2}. The input concentrations are set to $C_{b1} = 24.9 mol/cm^3$ and $C_{b2} = 0.1 mol/cm^3$. The constants associated with the rate of consumption are $k_1 = 1$ and $k_2 = 1$. The objective of the controller is to maintain the product concentration by adjusting the flow $w_2(t)$. To simplify the demonstration, we set $w_1(t) = 0.1 cm^3/s$. The level of the tank $h(t)$ is not controlled for this experiment.

To obtain the model reference controller for the CSTR, the plant model was trained using a sample time of $ts = 0.05$. We found that normalization was critical to obtaining a good controller. Controllers trained without normalized data were unsuccessful. Another important factor for the accurate training of the controller was considering the training modifications suggested in Chapter 6. The model reference controller requires dynamic training because of its recurrent architecture. The initial training resulted in saturation, oscillation and bad performance from the controller. As detailed in Chapter 6, the error surface of a recurrent network has spurious minima that occur in narrow valleys. These valleys can be mitigated by switching training sequences, using small random initial conditions and employing regularization. Using these techniques, we were able to obtain the controller response shown in Figure 7.4.

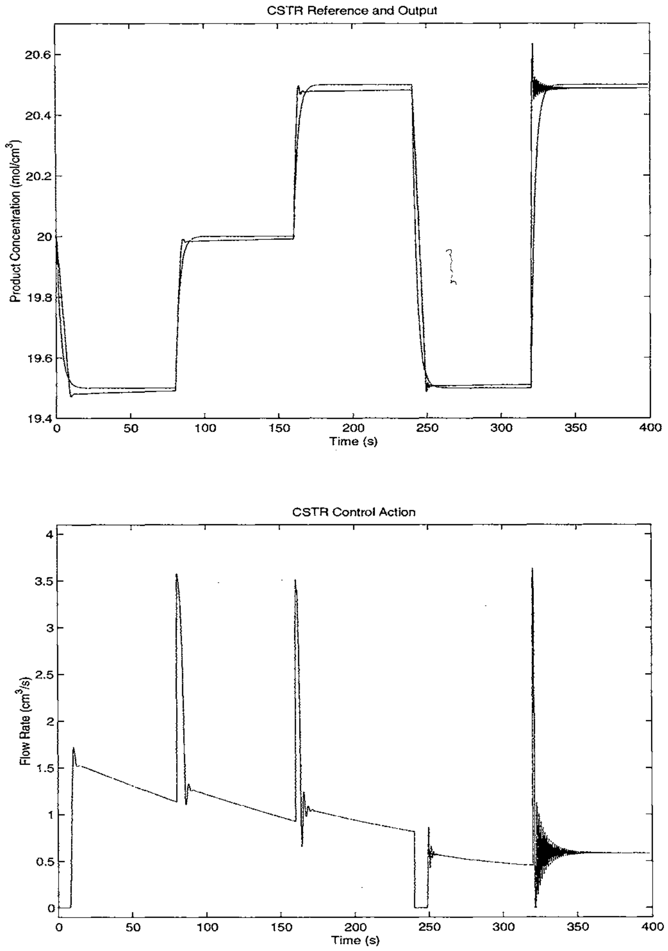

Figure 7.4: *CSTR response and control action using the Model Reference Controller.*

7.1.1.2. *Magnetic Levitation System (MagLev)*

In the second test problem, the objective is to control the position of a magnet suspended above an electromagnet, where the magnet is constrained so that it can only move in the vertical direction. The equation of motion is:

$$\frac{d^2y(t)}{dt^2} = -g + \operatorname{sgn}(i(t))\frac{\alpha i^2(t)}{My(t)} - \frac{\beta}{M}\frac{dy(t)}{dt} \qquad \textbf{(7.3)}$$

where $y(t)$ is the distance of the magnet above the electromagnet, $i(t)$ is the current flowing in the electromagnet, M is the mass of the magnet, and g is the gravitational constant. The parameter β is a viscous friction coefficient that is determined by the material in which the magnet moves, and α is a field strength constant that is determined by the number of turns of wire on the electromagnet and the strength of the magnet. The system is shown in Figure 7.5.

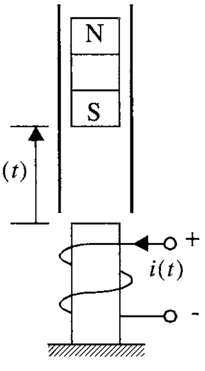

Figure 7.5: *Magnetic Levitation System*

Figure 7.6 shows the system response and the control actions for the system after training using the model reference controller. We see that the system has a steady state error of between 3% and 6%.

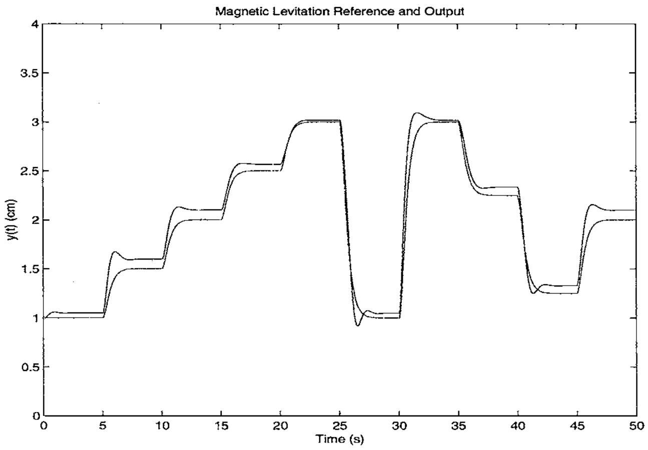

Figure 7.6: *MagLev response using model reference control.*

7.1.1.3. *Robot Arm*

The objective in this application is to control the movement of a simple, single-link robot arm (see Figure 7.7), where the equation of motion for the arm is:

$$\frac{d^2\phi}{dt^2} = -10\sin\phi - 2\frac{d\phi}{dt} + u \qquad \textbf{(7.4)}$$

where ϕ is the angle of the arm, and u is the torque supplied by the DC motor.

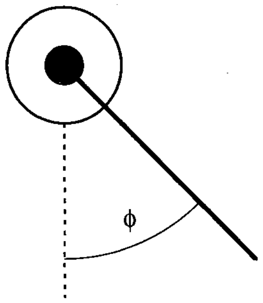

Figure 7.7: *Single-Link Robot Arm*

Figure 7.8 shows the response of the system using the model reference controller. The system is able to follow the reference, and the control actions are smooth.

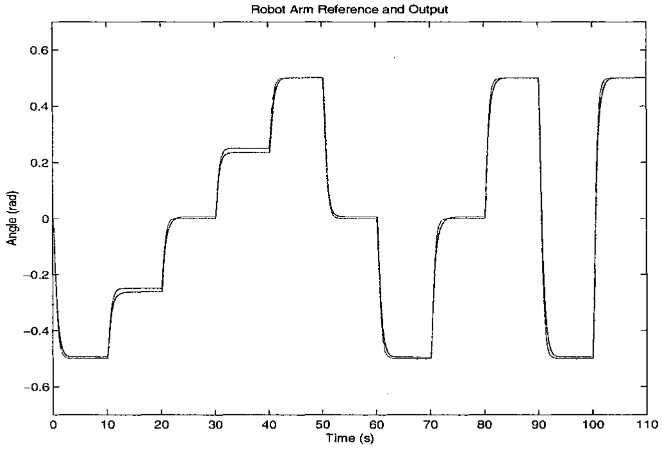

Figure 7.8: *Robot arm response.*

7.1.1.4. *Simple Diesel Engine Model*

The objective of this application is to control the speed of a diesel engine [7] by adjusting the fueling. The fueling has a variable time delay as a function of the engine speed. That delay can vary from 10 to 50 milliseconds and is reduced by the load applied to the engine. The effect of fueling $F_d(s)$ is modeled by the following equation:

$$N(s) = \frac{60}{2 \cdot \pi \cdot I \cdot s} F_d(s) \qquad (7.5)$$

where the engine inertia is $I = 2\ lb\text{-}ft\text{-}sec^2$, and $N(s)$ is the engine speed. A constant feedback gain of 0.2

was introduced to simulate viscous friction. The delay in the fuel system is $d(t) = 0.003 + 29.166 \cdot N_s(t)$, where $N_s(t)$ is a saturated engine speed between 500 and 2400 rpm. Figure 7.9 shows a Simulink representation of the simple engine model.

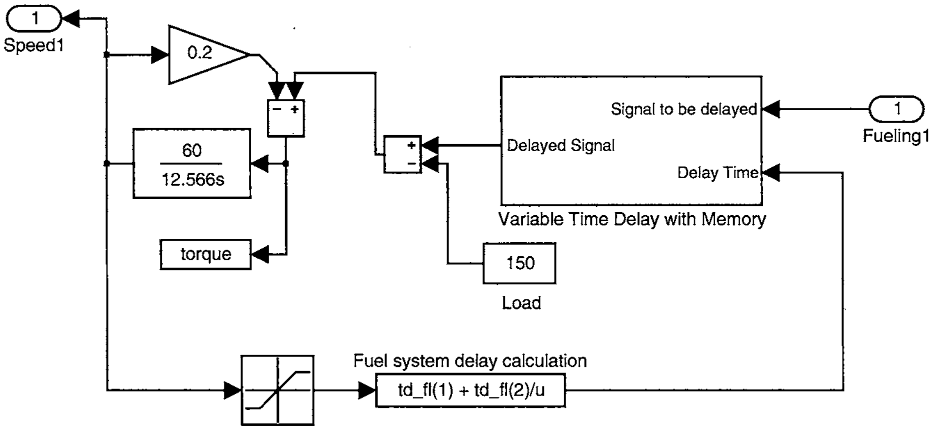

Figure 7.9: *Simple Engine Model.*

Figure 7.10 shows the engine speed for the model reference controller. The model reference controllers was able to control the engine including the effect of the variable delay in the fueling as shown in the next figure.

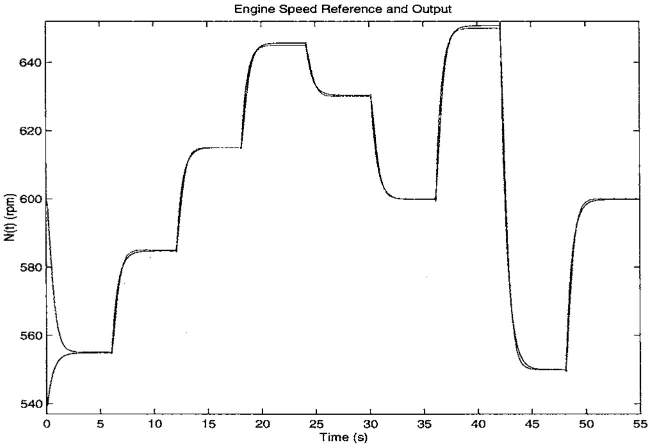

Figure 7.10: *Engine speed using the model reference controller and Predictive Controller.*

7.2. Filter design.

This section provides a second example of the application of the LDDN and dynamic backpropagation. We use a multi-loop recurrent network to predict the Mackey-Glass chaotic time series [28, 27]. This model was derived as a model of blood production represented by the equation:

$$\frac{du(t)}{dt} = \frac{au(t-\tau)}{1+u(t-\tau)^{10}} - bu(t) \qquad \textbf{(7.6)}$$

where $a = 0.2$, $b = 0.1$ and $u(t)$ represents the blood concentration at time t. For some patients with different pathologies the time τ increases and the blood concentration becomes chaotic for $\tau \geq 17$. Figure 7.11 shows the training data for $\tau = 17$ and $u(0) = 0.8$.

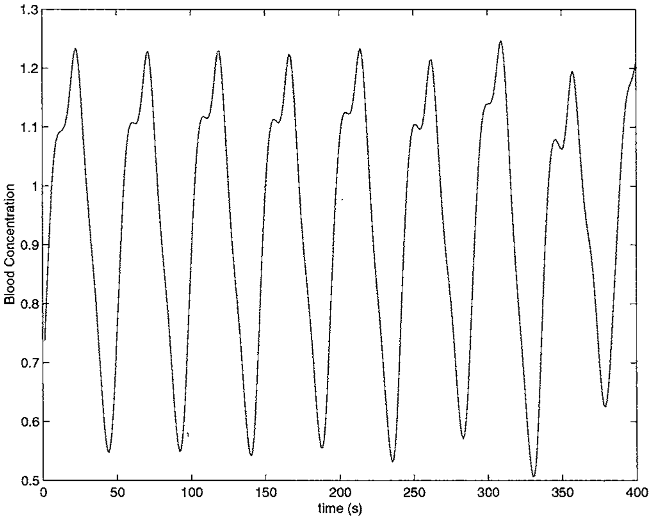

Figure 7.11: *Blood Concentration using the Mackey-Glass equation.*

The dynamic nature of the previous system makes the use of recurrent filter structures of great interest for prediction and control of this type of system. In the first part of this section we develop the dynamic training equations for the Mackey-Glass system. Then we present experimental results showing the prediction performance.

Figure 7.12 shows the structure of the LDRN used for predicting the Mackey-Glass chaotic time series. This is a modification proposed by Hagan et. al [19] of the cascaded recurrent neural network presented by Li and Haykin [27]. In this network there are 3 cascaded recurrent structures with the output connected to layer 6. We now show how these equations can be developed using our general BTT procedure, which was described in the flowchart of Chapter 5. The Backpropagation order is 6-5-4-3-2-1, so we start in the last layer in the backpropagation order (Layer 6) to get the following equations:

$$\mathbf{S}^{6,6}(t) = \dot{\mathbf{F}}^{6}(\mathbf{n}^{6}(t))\,;\; U' = \{6\}\;\;;\; E_S(6) = 6$$

$$\mathbf{S}^{6,5}(t) = \mathbf{S}^{6,6}(t)\mathbf{LW}^{6,5}(0)\dot{\mathbf{F}}^{5}(\mathbf{n}^{5}(t))$$

$$\mathbf{S}^{4,4}(t) = \dot{\mathbf{F}}^{4}(\mathbf{n}^{4}(t))\,;\; U' = \{6,4\}\;\;;\; E_S(4) = 4$$

$$\mathbf{S}^{4,3}(t) = \mathbf{S}^{4,4}(t)\mathbf{LW}^{4,3}(0)\dot{\mathbf{F}}^{3}(\mathbf{n}^{3}(t))$$

$$\mathbf{S}^{2,2}(t) = \dot{\mathbf{F}}^{2}(\mathbf{n}^{2}(t))\,;\; U' = \{6,4,2\}\;\;;\; E_S(2) = 2$$

$$\mathbf{S}^{2,1}(t) = \mathbf{S}^{2,2}(t)\mathbf{LW}^{2,1}(0)\dot{\mathbf{F}}^{1}(\mathbf{n}^{1}(t))$$

To compute the dynamic derivatives of the performance function with respect to the outputs of the system, we found that the only explicit derivative is with respect to output 6

$$\frac{\partial^e F}{\partial \mathbf{a}^6(t)} = -2(\mathbf{p}(t) - \mathbf{a}^6(t))$$

So the dynamic equations are:

$$\frac{\partial F}{\partial \mathbf{a}^{6}(t)} = \frac{\partial^e F}{\partial \mathbf{a}^{6}(t)} + \sum_{d \in DL_{5,6}} \mathbf{LW}^{5,6}(d)^T \mathbf{S}^{6,5}(t+d)^T \frac{\partial F}{\partial \mathbf{a}^{6}(t+d)}$$

$$\frac{\partial F}{\partial \mathbf{a}^{4}(t)} = \sum_{d \in DL_{3,4}} \mathbf{LW}^{3,4}(d)^T \mathbf{S}^{4,3}(t+d)^T \frac{\partial F}{\partial \mathbf{a}^{4}(t+d)} + \sum_{d \in DL_{5,4}} \mathbf{LW}^{5,4}(d)^T \mathbf{S}^{6,5}(t+d)^T \frac{\partial F}{\partial \mathbf{a}^{6}(t+d)}$$

$$\frac{\partial F}{\partial \mathbf{a}^{2}(t)} = \sum_{d \in DL_{1,2}} \mathbf{LW}^{1,2}(d)^T \mathbf{S}^{2,1}(t+d)^T \frac{\partial F}{\partial \mathbf{a}^{2}(t+d)} + \sum_{d \in DL_{3,2}} \mathbf{LW}^{3,2}(d)^T \mathbf{S}^{4,3}(t+d)^T \frac{\partial F}{\partial \mathbf{a}^{4}(t+d)}$$

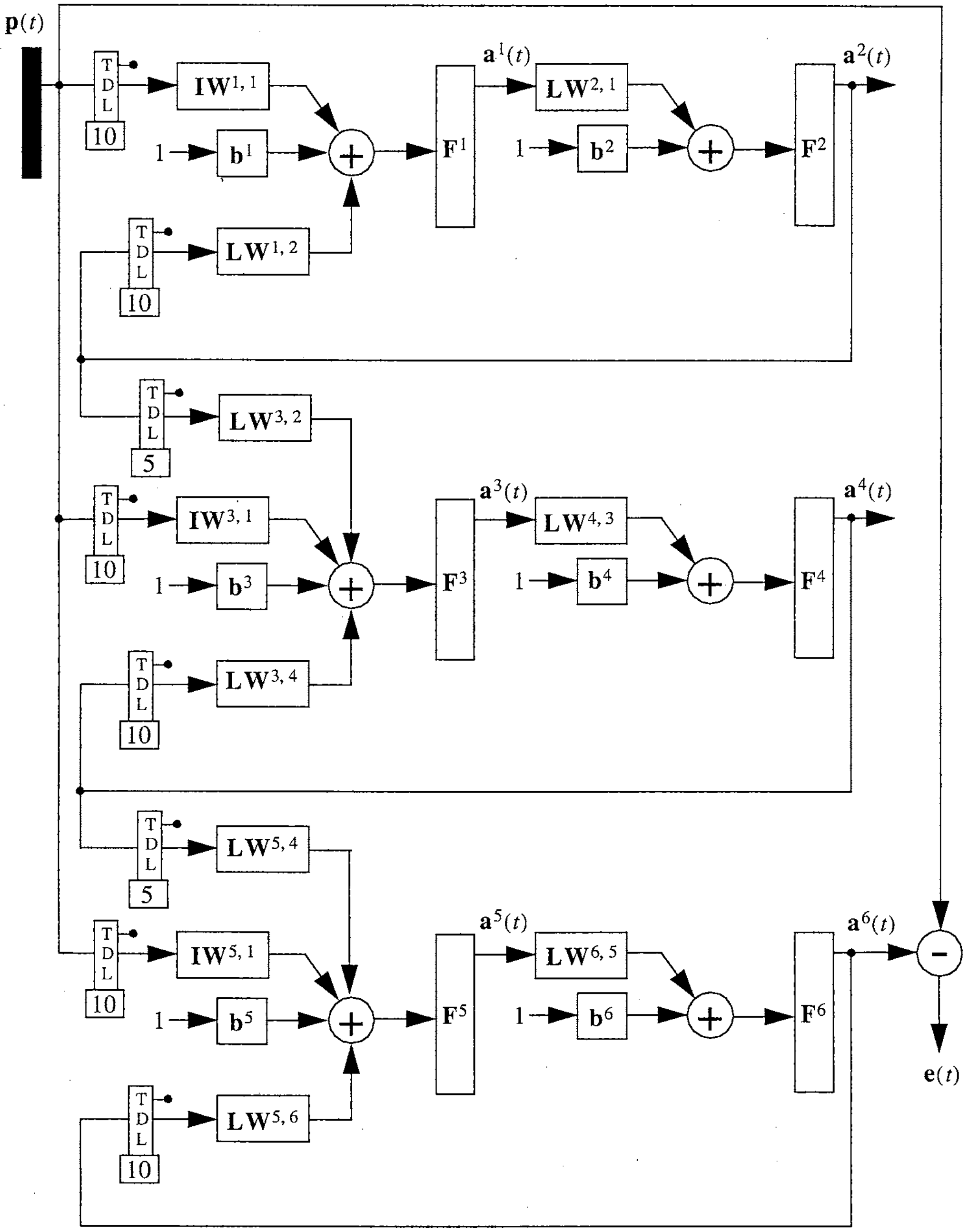

Figure 7.12: *Modified Cascaded Recurrent Neural Network*

For all layers we have

$$\mathbf{d}^6(t) = [\mathbf{S}^{6,6}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^6(t)}\ ;\ \mathbf{d}^5(t) = [\mathbf{S}^{6,5}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^6(t)}\ ;\ \mathbf{d}^4(t) = [\mathbf{S}^{4,4}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^4(t)}$$

$$\mathbf{d}^3(t) = [\mathbf{S}^{4,3}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^4(t)} ; \; \mathbf{d}^2(t) = [\mathbf{S}^{2,2}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^2(t)} ; \; \mathbf{d}^1(t) = [\mathbf{S}^{2,1}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^2(t)}$$

The previous process is repeated for each sample time in the training set.

Finally we can obtain the gradient for each weight and bias by

$$\frac{\partial F}{\partial \mathbf{b}^6} = \sum_{t=1}^{Q} \mathbf{d}^6(t) ; \; \frac{\partial F}{\partial \mathbf{LW}^{6,5}(d)} = \sum_{t=1}^{Q} \mathbf{d}^6(t) \times [\mathbf{a}^5(t-d)]^T ; \; \frac{\partial F}{\partial \mathbf{b}^5} = \sum_{t=1}^{Q} \mathbf{d}^5(t)$$

$$\frac{\partial F}{\partial \mathbf{IW}^{5,1}(d)} = \sum_{t=1}^{Q} \mathbf{d}^5(t) \times [\mathbf{p}^1(t-d)]^T ; \; \frac{\partial F}{\partial \mathbf{LW}^{5,4}(d)} = \sum_{t=1}^{Q} \mathbf{d}^5(t) \times [\mathbf{a}^4(t-d)]^T$$

$$\frac{\partial F}{\partial \mathbf{LW}^{5,6}(d)} = \sum_{t=1}^{Q} \mathbf{d}^5(t) \times [\mathbf{a}^6(t-d)]^T ; \; \frac{\partial F}{\partial \mathbf{b}^4} = \sum_{t=1}^{Q} \mathbf{d}^4(t) ; \frac{\partial F}{\partial \mathbf{LW}^{4,3}(d)} = \sum_{t=1}^{Q} \mathbf{d}^4(t) \times [\mathbf{a}^3(t-d)]^T$$

$$\frac{\partial F}{\partial \mathbf{b}^3} = \sum_{t=1}^{Q} \mathbf{d}^3(t) ; \; \frac{\partial F}{\partial \mathbf{IW}^{3,1}(d)} = \sum_{t=1}^{Q} \mathbf{d}^3(t) \times [\mathbf{p}^1(t-d)]^T ; \; \frac{\partial F}{\partial \mathbf{LW}^{3,2}(d)} = \sum_{t=1}^{Q} \mathbf{d}^3(t) \times [\mathbf{a}^2(t-d)]^T$$

$$\frac{\partial F}{\partial \mathbf{LW}^{3,4}(d)} = \sum_{t=1}^{Q} \mathbf{d}^3(t) \times [\mathbf{a}^4(t-d)]^T ; \; \frac{\partial F}{\partial \mathbf{b}^2} = \sum_{t=1}^{Q} \mathbf{d}^2(t) ; \frac{\partial F}{\partial \mathbf{LW}^{2,1}(d)} = \sum_{t=1}^{Q} \mathbf{d}^2(t) \times [\mathbf{a}^1(t-d)]^T$$

$$\frac{\partial F}{\partial \mathbf{b}^1} = \sum_{t=1}^{Q} \mathbf{d}^1(t) ; \; \frac{\partial F}{\partial \mathbf{IW}^{1,1}(d)} = \sum_{t=1}^{Q} \mathbf{d}^1(t) \times [\mathbf{p}^1(t-d)]^T ; \; \frac{\partial F}{\partial \mathbf{LW}^{1,2}(d)} = \sum_{t=1}^{Q} \mathbf{d}^1(t) \times [\mathbf{a}^2(t-d)]^T$$

After the network was trained, it was used to predict blood concentration. Figure 7.13 shows the actual and predicted signals when full dynamic backpropagation is used to train the LDDN. Figure 7.14 is a plot of the errors between actual and predicted signals.

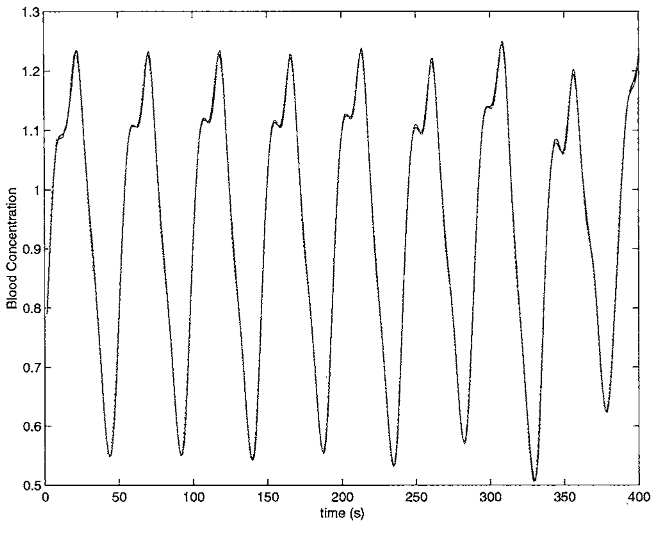

Figure 7.13: *Prediction Results for LDDN with Full Dynamic Training*

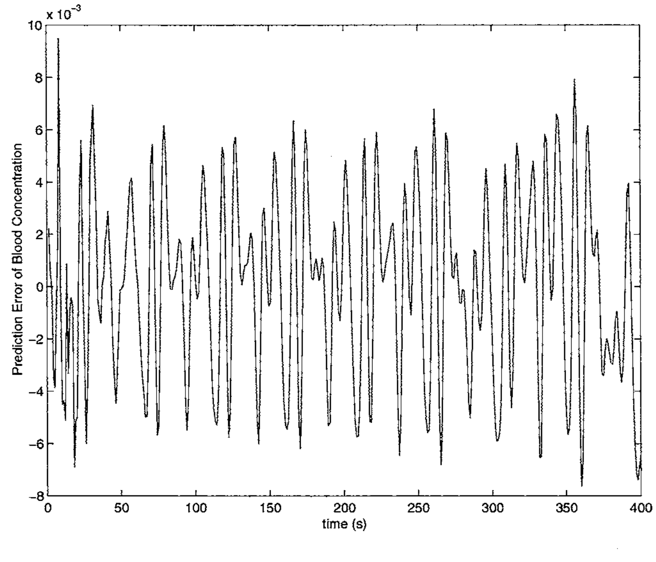

Figure 7.14: *Errors for LDDN with Full Dynamic Training*

Appendix C shows a modification of the previous dynamic neural network using the full cascaded recurrent neural network presented by Li and Haykin [27]. That appendix shows how the equations change for a similar architecture and how the algorithm is able to find the gradient calculation equations.

Chapter 8

ALGORITHM COMPARISON

This chapter will present memory, time and floating point operation comparisons for the gradient and Jacobian FP and BTT algorithm implementations using Matlab. The algorithms were tested for 33 different dynamic neural networks. The diagrams of each network are shown in Appendix D. All tests were performed in a Windows 2000 Professional system with a Pentium 4 running at 1.7 GHz and 640 MB of system RAM.

8.1. Memory requirements.

Table 8.1 shows the memory usage for the FP and BTT gradient algorithms. For each test network, the memory usage, the ratio of BTT memory to FP memory, the total number of weights and biases, and the total number of layers are presented. Except for two cases (network 32 and network 33), the BTT algorithm requires about 10 percent more memory than the FP algorithm. In the cases where the FP algorithm required more memory, the network had more weights because there were more delays in the network structure. Table 8.2 shows the memory usage for the FP and BTT Jacobian algorithms. For each test network, the Jacobian BTT implementation requires about twice as much memory as the FP algorithm.

Figure 8.1 and Figure 8.2 show the memory per weight required for the gradient and Jacobian algorithms. Networks with few weights need more memory per weight due to the baseline memory required to run the algorithm. Also, the BTT algorithm generally requires more memory per weight than the FP algorithm.

Table 8.1: *FP and BTT Gradient memory comparison (bytes).*

Test	Total Memory			Weights		Memory per Weight		
Network	FP	BTT	BTT/FP		Layers	FP	BTT	W/L
1	71952	85514	1.188	13	2	5535	6578	6.5
2	102522	118050	1.151	31	3	3307	3808	10.3
3	109688	121030	1.103	24	3	4570	5043	8.0
4	134506	153006	1.138	40	3	3363	3825	13.3
5	153444	171806	1.120	72	3	2131	2386	24.0
6	136246	152646	1.120	29	3	4698	5264	9.7
7	162296	179286	1.105	35	3	4637	5122	11.7
8	160150	177430	1.108	17	4	9421	10437	4.3
9	147948	165830	1.121	14	4	10568	11845	3.5
10	107614	125338	1.165	31	2	3471	4043	15.5
11	105920	117638	1.111	21	3	5044	5602	7.0
12	160440	178862	1.115	38	3	4222	4707	12.7
13	159296	179318	1.126	41	3	3885	4374	13.7
14	170728	190454	1.116	44	3	3880	4329	14.7
15	114474	133790	1.169	16	3	7155	8362	5.3
16	94100	111754	1.188	22	2	4277	5080	11.0
17	161802	190726	1.179	38	4	4258	5019	9.5
18	189238	210662	1.113	17	5	11132	12392	3.4
19	71952	85514	1.188	13	2	5535	6578	6.5
20	165554	187294	1.131	12	5	13796	15608	2.4
21	375912	404158	1.075	32	8	11747	12630	4.0
22	197042	223042	1.132	19	6	10371	11739	3.2
23	141256	161910	1.146	32	4	4414	5060	8.0
24	186932	215388	1.152	40	5	4673	5385	8.0
25	346856	392178	1.131	85	9	4081	4614	9.4
26	47674	54474	1.143	3	1	15891	18158	3.0
27	126964	155954	1.228	32	3	3968	4874	10.7
28	454060	484850	1.068	81	10	5606	5986	8.1
29	259240	301002	1.161	51	6	5083	5902	8.5
30	48362	55098	1.139	4	1	12091	13775	4.0
31	83538	98738	1.182	19	2	4397	5197	9.5
32	318182	286450	0.900	101	6	3150	2836	16.8
33	433800	366510	0.845	115	7	3772	3187	16.4

Table 8.2: *FP and BTT Jacobian memory comparison (bytes).*

Test	Total Memory			Weights		Memory per Weight		
Network	FP	BTT	BTT/FP		Layers	FP	BTT	W/L
1	78534	191894	2.443	13	2	6041	14761	6.5
2	113956	231130	2.028	31	3	3676	7456	10.3
3	121730	245686	2.018	24	3	5072	10237	8.0
4	150652	348006	2.310	40	3	3766	8700	13.3
5	176126	320350	1.819	72	3	2446	4449	24.0
6	154064	283078	1.837	29	3	5313	9761	9.7
7	182698	374126	2.048	35	3	5220	10689	11.7
8	178412	310514	1.740	17	4	10495	18266	4.3
9	164082	287218	1.750	14	4	11720	20516	3.5
10	121948	258910	2.123	31	2	3934	8352	15.5
11	117506	188846	1.607	21	3	5596	8993	7.0
12	181298	374158	2.064	38	3	4771	9846	12.7
13	180610	375070	2.077	41	3	4405	9148	13.7
14	192498	391078	2.032	44	3	4375	8888	14.7
15	126972	288590	2.273	16	3	7936	18037	5.3
16	105394	222846	2.114	22	2	4791	10129	11.0
17	179912	364074	2.024	38	4	4735	9581	9.5
18	209920	387198	1.845	17	5	12348	22776	3.4
19	78534	191894	2.443	13	2	6041	14761	6.5
20	182132	346174	1.901	12	5	15178	28848	2.4
21	479878	1137402	2.370	32	8	14996	35544	4.0
22	213912	384062	1.795	19	6	11259	20214	3.2
23	156782	280930	1.792	32	4	4899	8779	8.0
24	270594	630768	2.331	40	5	6765	15769	8.0
25	446218	816954	1.831	85	9	5250	9611	9.4
26	52444	105450	2.011	3	1	17481	35150	3.0
27	140222	281946	2.011	32	3	4382	8811	10.7
28	978386	1429918	1.462	81	10	12079	17653	8.1
29	502194	864926	1.722	51	6	9847	16959	8.5
30	291006	532358	1.829	4	1	72752	133090	4.0
31	53284	110626	2.076	19	2	2804	5822	9.5
32	92704	207702	2.240	101	6	918	2056	16.8
33	355876	614310	1.726	115	7	3095	5342	16.4

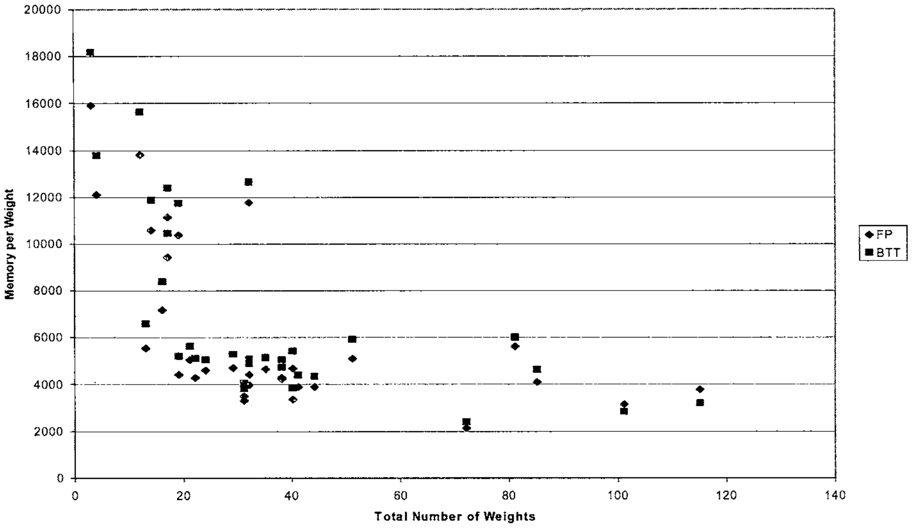

Figure 8.1: *Memory per Weight for gradient algorithms.*

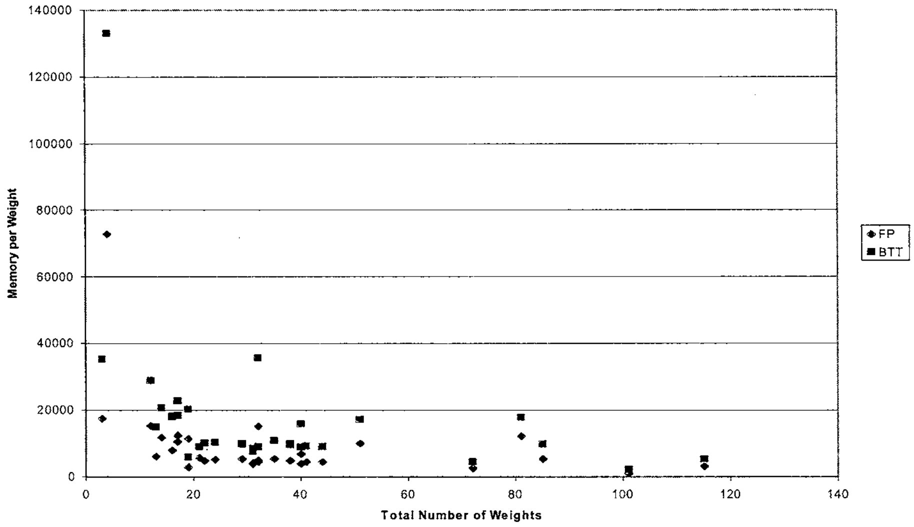

Figure 8.2: *Memory per Weight for Jacobian algorithms.*

Table 8.3 and Table 8.4 show the memory usage for network 1 (Figure D.1) and network 31 (Figure D.31) as the size of the first layer is increased. The first network has feedforward delays. The second network has feedback delays from layer 1 to itself. Both networks have two layers. From the tables, we see that the memory for the BTT gradient algorithm increases faster than the FP gradient algorithm as the size of the first layer increases. A different behavior is seen for the Jacobian algorithms, where the FP memory requirements increase faster than BTT as the the size of the first layer increases.

Table 8.3: *FP and BTT memory comparison (bytes) for network 1 as function of first layer size.*

	Gradient			Jacobian			
S1	FP	BTT	BTT/FP	FP	BTT	BTT/FP	Weights
1	67276	75698	1.125	72642	173470	2.388	5
3	72940	85858	1.177	79522	192238	2.417	13
5	81740	101522	1.242	89538	216510	2.418	21
7	93676	122690	1.310	102690	246286	2.398	29
9	108748	149362	1.373	118978	281566	2.367	37
11	126956	181538	1.430	138402	322350	2.329	45
13	148300	219218	1.478	160962	368638	2.290	53

Table 8.4: *FP and BTT memory comparison (bytes) for network 31 as function of first layer size.*

	Gradient			Jacobian			
S1	FP	BTT	BTT/FP	FP	BTT	BTT/FP	Weights
1	76056	85998	1.131	83094	185442	2.232	5
2	79688	91678	1.150	87638	195730	2.233	11
3	84744	99182	1.170	93910	208146	2.216	19
4	91320	108510	1.188	102006	222690	2.183	29
5	99512	119662	1.202	112022	239362	2.137	41
6	109416	132638	1.212	124054	258162	2.081	55

Table 8.5 and Table 8.6 show the memory usage for the network 1 (Figure D.1) and network 31 (Figure D.31) as the number of delays is increased. These tables show a different pattern than was seen for layer size. In this case, the memory requirements for the BTT gradient increase slower than those for the FP gradient as the number of delays increases. On the other hand, the memory requirements for the BTT Jacobian increase faster than those for the FP Jacobian.

Table 8.5: *FP and BTT memory comparison (bytes) for network 1 as function of number of delays.*

	Gradient			Jacobian			
ND	FP	BTT	BTT/FP	FP	BTT	BTT/FP	Weights
1	72940	85858	1.177	79522	192238	2.417	13
3	77836	89506	1.150	85330	216238	2.534	19
5	83500	93154	1.116	91906	241966	2.633	25
7	89932	96802	1.076	99250	269422	2.715	31
9	97132	100450	1.034	107362	298606	2.781	37
11	105100	104098	0.990	116242	329518	2.835	43
13	113836	107746	0.947	125890	362158	2.877	49
15	123340	111394	0.903	136306	396526	2.909	55

Table 8.6: *FP and BTT memory comparison (bytes) for network 31 as function of number of delays.*

	Gradient			Jacobian			
ND	FP	BTT	BTT/FP	FP	BTT	BTT/FP	Weights
1	84744	99182	1.170	93910	208146	2.216	19
2	88472	101414	1.146	99006	221250	2.235	28
3	92776	103646	1.117	104678	234786	2.243	37
4	97656	105878	1.084	110926	248754	2.243	46
5	103112	108110	1.048	117750	263154	2.235	55
6	109144	110342	1.011	125150	277986	2.221	64

Table 8.7 and Table 8.8 shown the memory usage for the network 1 (Figure D.1) and network 31 (Figure D.31) as the number of samples of the training sequence is increased. These tables shown that the memory requirements for the BTT gradient and Jacobian increase faster than those for the FP gradient and Jacobian as the number of samples increases.

Table 8.7: *FP and BTT memory comparison (bytes) for network 1 as function of number of samples.*

	Gradient			Jacobian		
NS	FP	BTT	BTT/FP	FP	BTT	BTT/FP
20	79662	91330	1.146	87164	218070	2.502
40	106702	133650	1.253	123164	558790	4.537
60	133742	175970	1.316	159164	1072310	6.737
80	160782	218290	1.358	195164	1758630	9.011
100	187822	260610	1.388	231164	2617750	11.324
120	214862	302930	1.410	267164	3649670	13.661
140	241902	345250	1.427	303164	4854390	16.012
160	268570	387570	1.443	339164	6231910	18.374
180	295890	429890	1.453	375164	7782230	20.744
200	323022	472210	1.462	411164	9505350	23.118
220	350062	514530	1.470	447164	11401270	25.497
240	377102	556850	1.477	483164	13469990	27.879
260	404142	559170	1.384	519164	(*)	(*)
280	431182	641490	1.488	555164	(*)	(*)
300	458222	683810	1.492	591164	(*)	(*)
400	593422	895410	1.509	771164	(*)	(*)
500	728622	1107010	1.519	951164	(*)	(*)
600	863822	1318610	1.526	1131164	(*)	(*)
1000	1404622	2165010	1.541	1851164	(*)	(*)
5000	6812622	10629010	1.560	9051164	(*)	(*)

(*) Tests were not performed due to computer memory limitations.

Table 8.8: *FP and BTT memory comparison (bytes) for network 31 as function of number of samples.*

	Gradient			Jacobian		
NS	FP	BTT	BTT/FP	FP	BTT	BTT/FP
20	94770	105638	1.115	106680	236786	2.220
40	128210	156598	1.221	153720	590786	3.843
60	161650	207558	1.284	200760	1117586	5.567
80	195090	258518	1.325	247800	1817186	7.333
100	228530	309478	1.354	294840	2689586	9.122
120	261970	360438	1.376	341880	3734786	10.924
140	295410	411398	1.393	388920	4952786	12.735
160	328850	462358	1.406	435960	6343586	14.551
180	362290	513318	1.417	483000	7907186	16.371
200	395730	564278	1.426	530040	9643586	18.194
220	429170	615238	1.434	577080	11552786	20.019
240	462610	666198	1.440	624120	13634786	21.846
260	496050	717158	1.446	671160	(*)	(*)
280	529490	768118	1.451	718200	(*)	(*)
300	562930	819078	1.455	795240	(*)	(*)
400	760130	1073878	1.413	1000440	(*)	(*)
500	897330	1328678	1.481	1235640	(*)	(*)
600	1064530	1583478	1.487	1470840	(*)	(*)
1000	1733330	2602678	1.502	2413330	(*)	(*)
5000	8421330	12794678	1.519	11819640	(*)	(*)

(*) Tests were not performed due to computer memory limitations.

8.2. Speed Comparison.

Table 8.9 shows the average time required to compute the gradient for each one of the test networks using FP and BTT. The BTT gradient is between 1.7 and 3.8 times faster than the FP gradient. Table 8.10 shows a similar table for the FP and BTT Jacobian. Here the FP Jacobian is between 1.2 and 4.2 times faster than the BTT Jacobian.

Table 8.9: *FP and BTT Gradient time comparison.*

Test	Total Time (ms)			Weights		Time per Weight (ms)		
Network	FP	BTT	BTT/FP		Layers	FP	BTT	W/L
1	394.7	186.4	0.472	13	2	30.4	14.3	6.5
2	570.3	258.5	0.453	31	3	18.4	8.3	10.3
3	761.8	347.7	0.456	24	3	31.7	14.5	8.0
4	1173.5	515.6	0.439	40	3	29.3	12.9	13.3
5	1240.9	513.5	0.414	72	3	17.2	7.1	24.0
6	1429.9	577.8	0.404	29	3	49.3	19.9	9.7
7	2817.9	958.3	0.340	35	3	80.5	27.4	11.7
8	1705.1	652.4	0.383	17	4	100.3	38.4	4.3
9	1350.9	559.7	0.414	14	4	96.5	40.0	3.5
10	937.7	428.1	0.457	31	2	30.2	13.8	15.5
11	593.9	286.5	0.482	21	3	28.3	13.6	7.0
12	2369.4	839.0	0.354	38	3	62.4	22.1	12.7
13	1952.3	755.8	0.387	41	3	47.6	18.4	13.7
14	2877.3	965.3	0.335	44	3	65.4	21.9	14.7
15	1078.1	468.5	0.435	16	3	67.4	29.3	5.3
16	838.1	356.9	0.426	22	2	38.1	16.2	11.0
17	1812.8	677.5	0.374	38	4	47.7	17.8	9.5
18	2619.5	888.0	0.339	17	5	154.1	52.2	3.4
19	398.2	187.9	0.472	13	2	30.6	14.5	6.5
20	1957.9	725.6	0.371	12	5	163.2	60.5	2.4
21	8902.1	2411.5	0.271	32	8	278.2	75.4	4.0
22	2180.5	817.2	0.375	19	6	114.8	43.0	3.2
23	972.3	413.3	0.425	32	4	30.4	12.9	8.0
24	1271.9	514.6	0.405	40	5	31.8	12.9	8.0
25	3095.4	1113.4	0.360	85	9	36.4	13.1	9.4
26	225.1	119.9	0.533	3	1	75.0	40.0	3.0
27	917.5	374.6	0.408	32	3	28.7	11.7	10.7
28	9181.3	2367.4	0.258	81	10	113.3	29.2	8.1
29	8303.0	2244.2	0.270	51	6	162.8	44.0	8.5
30	4286.0	1258.6	0.294	4	1	1071.5	314.7	4.0
31	246.5	137.1	0.556	19	2	13.0	7.2	9.5
32	598.6	275.9	0.461	101	6	5.9	2.7	16.8
33	4018.7	1398.8	0.348	115	7	34.9	12.2	16.4

Table 8.10: *FP and BTT Jacobian time comparison.*

Test	Total Time (ms)			Weights		Time per Weight (ms)		
Network	FP	BTT	BTT/FP		Layers	FP	BTT	W/L
1	395.7	980.3	2.477	13	2	30.4	75.4	6.5
2	572.8	1181.7	2.063	31	3	18.5	38.1	10.3
3	780.2	1858.9	2.383	24	3	32.5	77.5	8.0
4	1183.7	2985.2	2.522	40	3	29.6	74.6	13.3
5	1254.7	2886.6	2.301	72	3	17.4	40.1	24.0
6	1436.8	3338.1	2.323	29	3	49.5	115.1	9.7
7	2839.9	6400.6	2.254	35	3	81.1	182.9	11.7
8	1709.5	3491.2	2.042	17	4	100.6	205.4	4.3
9	1369.0	2791.1	2.039	14	4	97.8	199.4	3.5
10	944.5	2603.7	2.757	31	2	30.5	84.0	15.5
11	604.2	1291.9	2.138	21	3	28.8	61.5	7.0
12	2368.8	5461.6	2.306	38	3	62.3	143.7	12.7
13	1956.1	4558.8	2.331	41	3	47.7	111.2	13.7
14	2890.8	6423.5	2.222	44	3	65.7	146.0	14.7
15	1090.5	2611.9	2.395	16	3	68.2	163.2	5.3
16	843.7	2072.5	2.456	22	2	38.4	94.2	11.0
17	1835.0	3630.6	1.979	38	4	48.3	95.5	9.5
18	2640.8	4923.2	1.864	17	5	155.3	289.6	3.4
19	397.1	965.3	2.431	13	2	30.5	74.3	6.5
20	1952.3	3614.8	1.852	12	5	162.7	301.2	2.4
21	9358.7	25048.0	2.676	32	8	292.5	782.8	4.0
22	2196.9	3925.7	1.787	19	6	115.6	206.6	3.2
23	978.5	1810.3	1.850	32	4	30.6	56.6	8.0
24	1544.1	6627.6	4.292	40	5	38.6	165.7	8.0
25	3438.0	8609.3	2.504	85	9	40.4	101.3	9.4
26	230.3	599.9	2.605	3	1	76.8	200.0	3.0
27	935.9	1794.5	1.917	32	3	29.2	56.1	10.7
28	9193.8	11360.3	1.236	81	10	113.5	140.3	8.1
29	8282.0	10927.0	1.319	51	6	162.4	214.3	8.5
30	4317.1	6613.9	1.532	4	1	1079.3	1653.5	4.0
31	248.3	743.0	2.992	19	2	13.1	39.1	9.5
32	612.2	1403.0	2.292	101	6	6.1	13.9	16.8
33	4089.3	8985.9	2.197	115	7	35.6	78.1	16.4

Table 8.11 and Table 8.12 show the time required to compute the gradient and Jacobian for network 1 (Figure D.1) and network 31 (Figure D.31) as the size of the first layer is increased. The BTT gradient algorithm is approximately twice as fast as the FP gradient for all sizes. However, the FP Jacobian is approximately three times as fast as BTT Jacobian for all sizes.

Table 8.11: *FP and BTT time comparison for network 1 as function of first layer size (msec).*

	Gradient			Jacobian			
S1	FP	BTT	BTT/FP	FP	BTT	BTT/FP	Weights
1	147.4	78.3	0.531	140.7	452.0	3.213	5
3	143.0	77.7	0.543	146.4	459.6	3.139	13
5	144.7	76.1	0.526	148.3	465.4	3.138	21
7	149.1	80.4	0.539	150.2	475.6	3.166	29
9	153.8	84.9	0.552	154.7	484.4	3.131	37
11	158.7	87.5	0.551	158.5	485.8	3.065	45
13	165.0	92.8	0.562	167.6	495.6	2.957	53

Table 8.12: *FP and BTT time comparison for network 31 as function of first layer size (msec).*

	Gradient			Jacobian			
S1	FP	BTT	BTT/FP	FP	BTT	BTT/FP	Weights
1	220.1	109.1	0.496	225.7	712.5	3.157	5
2	226.8	112.0	0.494	236.1	735.8	3.116	11
3	230.1	116.5	0.506	239.4	740.5	3.093	19
4	235.2	116.8	0.497	240.9	743.9	3.088	29
5	243.7	120.3	0.494	245.0	751.5	3.067	41
6	257.6	126.7	0.492	260.5	771.0	2.960	55

Table 8.13 and Table 8.14 shown the time comparison as we increase the number of delays. The time between FP gradient and BTT gradient tends to get closer as the number of delays increases. On the other hand, the BTT Jacobian tends to get much slower than the FP Jacobian as the number of delays increases.

Table 8.13: *FP and BTT time comparison for network 1 as function of number of delays (msec).*

	Gradient			Jacobian			
ND	FP	BTT	BTT/FP	FP	BTT	BTT/FP	Weights
1	142.3	77.7	0.546	142.8	456.7	3.198	13
3	150.7	93.8	0.622	151.5	616.3	4.068	19
5	159.0	110.6	0.696	158.8	774.1	4.875	25
7	171.0	126.3	0.739	170.2	931.5	5.473	31
9	180.8	144.3	0.798	182.1	1105.9	6.073	37
11	194.8	161.4	0.829	198.1	1266.6	6.394	43
13	207.9	176.6	0.849	209.1	1426.6	6.823	49
15	223.2	195.6	0.876	226.9	1592.3	7.018	55

Table 8.14: *FP and BTT time comparison for network 31 as function of number of delays (msec).*

	Gradient			Jacobian			
ND	FP	BTT	BTT/FP	FP	BTT	BTT/FP	Weights
1	229.3	114.1	0.498	234.6	736.8	3.141	19
2	241.9	132.2	0.547	250.3	901.7	3.602	28
3	254.2	148.0	0.582	261.4	1079.2	4.129	37
4	271.7	168.7	0.621	276.7	1241.3	4.486	46
5	291.5	185.8	0.637	296.2	1412.5	4.769	55
6	312.5	203.7	0.652	317.6	1580.9	4.978	64

Table 8.15 and Table 8.16 shown the time comparison as we increase the number of samples of the training sequence. The time between FP gradient and BTT gradient tends to get closer as the number of samples increases. For a large number of samples, FP becomes a better option than BTT for computing the gradient. On the other hand, the BTT Jacobian tends to get much slower than the FP Jacobian as the number of samples increases.

Table 8.15: *FP and BTT time comparison for network 1 as function of number of samples (msec).*

	Gradient			Jacobian		
NS	FP	BTT	BTT/FP	FP	BTT	BTT/FP
20	163.7	140.0	0.855	150.5	360.5	2.395
40	280.0	147.0	0.525	285.5	1517.0	5.313
60	414.0	230.3	0.556	425.0	5092.5	11.982
80	557.7	310.3	0.556	566.0	13700.0	24.205
100	687.7	397.3	0.578	706.0	28251.0	40.016
120	828.0	494.3	0.597	836.5	47934.0	57.303
140	967.7	597.7	0.618	981.0	77956.5	79.466
160	1118.3	711.0	0.636	1146.0	121474.5	105.999
180	1252.0	828.0	0.661	1262.0	184525.5	146.217
200	1408.7	965.0	0.685	1447.0	270569.5	186.987
220	1532.3	1071.3	0.699	1552.5	378234.0	243.629
240	1676.0	1222.0	0.729	1687.0	523673.5	310.417
260	1829.0	1368.7	0.748	1838.0	(*)	(*)
280	1956.3	1515.7	0.775	1978.0	(*)	(*)
300	2093.0	1676.0	0.801	2113.0	(*)	(*)
400	2857.7	2814.0	0.985	2829.0	(*)	(*)
500	3495.0	3731.7	1.068	3535.5	(*)	(*)
600	4152.7	4957.0	1.194	4251.0	(*)	(*)
1000	7050.3	14040.3	1.991	7245.0	(*)	(*)
5000	40054.0	743108.7	18.553	46877.5	(*)	(*)

(*) Tests were not performed due to computer memory limitations.

Table 8.16: *FP and BTT time comparison for network 31 as function of number of samples (msec).*

	Gradient			Jacobian		
NS	FP	BTT	BTT/FP	FP	BTT	BTT/FP
20	230.3	110.0	0.478	230.5	596.0	2.586
40	471.0	240.0	0.510	465.5	2538.5	5.453
60	684.3	360.7	0.527	696.0	7736.0	11.115
80	911.3	501.0	0.550	926.5	18737.0	20.223
100	1141.7	657.7	0.576	1162.0	37138.5	31.961
120	1378.7	821.0	0.595	1392.0	67517.0	48.504
140	1609.0	1001.7	0.623	1617.5	111129.5	68.704
160	1849.3	1178.7	0.637	1893.0	167681.0	88.580
180	2069.7	1375.3	0.664	2078.0	233611.0	112.421
200	2355.2	1604.2	0.681	2409.0	329509.0	136.782
220	2540.0	1816.3	0.715	2618.5	448520.0	171.289
240	2764.0	2063.0	0.746	2789.0	608810.5	218.290
260	2991.0	2313.7	0.774	3019.5	(*)	(*)
280	3228.3	2584.0	0.800	3259.5	(*)	(*)
300	3465.0	2894.3	0.835	3500.0	(*)	(*)
400	4642.7	4649.7	1.002	4682.0	(*)	(*)
500	5818.0	6930.0	1.191	5868.0	(*)	(*)
600	6945.0	9724.0	1.400	7076.0	(*)	(*)
1000	11947.0	29422.0	2.463	12193.0	(*)	(*)
5000	78362.0	1488820.0	18.999	87185.5	(*)	(*)

(*) Tests were not performed due to computer memory limitations.

8.3. Computational complexity.

The complexities of the various algorithms are dependent on the number of weights in the network, N, and also on the length of the training sequence, T. In this section we will analyze experiments to verify the complexity of each algorithm in terms of both N and T. We begin with some background discussion of complexity that will help in understanding the tables that follows.

Let's say that the number of flops required for certain algorithm is $O(N^k)$:

$$flops = a_0 + a_1 N + a_2 N^2 + a_3 N^3 + \ldots + a_k N^k \tag{8.1}$$

As $N \to \infty$ this can be approximated as

$$flops \cong a_k N^k \tag{8.2}$$

If we take the log of the number of flops to the base N we have

$$\begin{aligned} \log_N flops &\cong \log_N a_k + k \log_N N \\ &\cong \log_N a_k + k \end{aligned} \tag{8.3}$$

As $N \to \infty$ this should approach k. If we know k, we can find a_k through the following

$$N^{(\log_N flops - k)} \cong a_k \tag{8.4}$$

If we know a_k, we can then check the value of k by

$$\log_N (flops / a_k) \cong k \tag{8.5}$$

We will be using Eq. (8.3), Eq. (8.4) and Eq. (8.5) in the tables in this section in order to experimentally verify the order of the complexity of the various algorithms.

As mentioned earlier, it is best to view the complexity of the algorithms to be functions of two variables: the number of weights in the network, N, and the number of time steps in the training sequence or samples, T. We will begin by testing the effect of N on algorithm complexity. We will do this in two ways: first, we will increase the number of neurons in the first layer of the network, then we will increase the number of delays in the network. Next, we will test the effect of T on algorithm complexity by holding the network architecture fixed while we increase the length of the training sequence.

Table 8.17 and Table 8.19 show the floating point operations required to compute the gradient and Jacobian for network 1 (Figure D.1) and network 31 (Figure D.31) as the size of the first layer is increased, for sequences of 20 samples and with the number of delays equal to 1. The BTT algorithm requires fewer computations than the FP algorithm for the gradient calculation, but more computations than the FP algorithm for the Jacobian calculations. As the size of the first layer increases, the number of flops for the BTT gradient increases slower than the number of flops for the FP gradient. For the Jacobian calculations, the number of flops for FP and BTT tend to get closer as the size of the first layer increases.

Next, let's examine these experiments in a more qualitative way, using Eq. (8.3), Eq. (8.4) and Eq. (8.5). Table 8.17 and Table 8.19 show $\log_N(flops)$ on the right side of the tables. From Eq. (8.3) this value should approach k as N increases.

In Table 8.18 and Table 8.20 we use Eq. (8.4) and Eq. (8.5). The columns labeled $F1(k)$ represent Eq. (8.4) and the columns labeled $F2$ represent Eq. (8.5). For example, the second column in Table 8.18 indicates that $a_k = 0.63$. Using this value in Eq. (8.5), we obtain column three in Table 8.18, which indicates that $k = 3$. This means that the FP algorithm for the gradient calculation is $O(N^3)$.

Summarizing the results from the tables, the data suggests that for network 1 the FP gradient and Jacobian algorithms are $O(N^3)$, and the BTT algorithms are $O(N^2)$. For network 31, the FP algorithms are $O(N^2)$ and the BTT algorithms are $O(N^{1.5})$. Network 31 contains feedback. while network 1 does not; this seems to explain the differences in algorithm complexity, and similar results were seen in other tests not shown here.

Table 8.17: *FP and BTT floating point operations for network 1 as function of first layer size.*

	Flops						$\log_N(Flops)$				
	Gradient			Jacobian			Gradient		Jacobian		
S1	FP	BTT	BTT/FP	FP	BTT	BTT/FP	FP	BTT	FP	BTT	Weights (N)
1	3782	1515	0.401	4917	18807	3.825	5.12	4.55	5.28	6.12	5
3	8802	3035	0.345	9923	34767	3.504	3.54	3.13	3.59	4.08	13
5	19402	5475	0.282	20523	60387	2.942	3.24	2.83	3.26	3.62	21
7	35522	8875	0.250	38643	96087	2.487	3.11	2.70	3.14	3.41	29
9	65082	13235	0.203	66203	141867	2.143	3.07	2.63	3.07	3.29	37
11	104014	18555	0.178	105123	197727	1.881	3.03	2.58	3.04	3.20	45
13	156202	24835	0.159	157323	263667	1.676	3.01	2.55	3.01	3.14	53
20	471112	54288	0.115	472323	573837	1.215	2.97	2.48	2.97	3.02	81
40	3131512	203488	0.065	3132723	2140437	0.683	2.94	2.41	2.94	2.87	161
60	9903912	448688	0.045	9905123	4715037	0.476	2.94	2.37	2.94	2.80	241
80	22708312	789888	0.035	22709523	8297637	0.365	2.93	2.35	2.93	2.76	321
100	43464712	1227088	0.028	43465923	12888240	0.297	2.93	2.34	2.93	2.73	401
200	333726712	4853088	0.015	333727923	50961237	0.153	2.94	2.30	2.94	2.65	801
300	1110788712	10879088	0.010	1110789923	114234237	0.103	2.94	2.28	2.94	2.62	1201
400	2614650712	19305088	0.007	2614651923	202707237	0.078	2.94	2.27	2.94	2.59	1601
500	5085312723	30131091	0.006	5085313923	316380237	0.062	2.94	2.27	2.94	2.57	2001
600	8762774712	43357088	0.005	8762775923	455253237	0.052	2.94	2.26	2.94	2.56	2401

Table 8.18: *FP and BTT constant calculations for floating point operations for network 1 as a function of first layer size.*

	FP Gradient		BTT Gradient		FP Jacobian		BTT Jacobian	
S1	F1(3)	F2	F1(2)	F2	F1(3)	F2	F1(2)	F2
1	30.26	5.406	60.60	3.298	39.34	5.569	752.28	3.400
3	4.01	3.721	17.96	2.340	4.52	3.768	205.72	2.373
5	2.10	3.395	12.41	2.166	2.22	3.413	136.93	2.181
7	1.46	3.249	10.55	2.101	1.58	3.274	114.25	2.110
9	1.28	3.197	9.67	2.070	1.31	3.202	103.63	2.075
11	1.14	3.156	9.16	2.053	1.15	3.159	97.64	2.056
13	1.05	3.128	8.84	2.041	1.06	3.130	93.87	2.043
20	0.89	3.078	8.27	2.022	0.89	3.078	87.46	2.023
40	0.75	3.034	7.85	2.009	0.75	3.034	82.58	2.009
60	0.71	3.021	7.73	2.005	0.71	3.021	81.18	2.005
80	0.69	3.015	7.67	2.004	0.69	3.015	80.53	2.003
100	0.67	3.011	7.63	2.003	0.67	3.011	80.15	2.002
200	0.65	3.005	7.56	2.001	0.65	3.005	79.43	2.001
300	0.64	3.002	7.54	2.001	0.64	3.002	79.20	2.000
400	0.64	3.002	7.53	2.001	0.64	3.002	79.08	2.000
500	0.63	3.001	7.53	2.000	0.63	3.001	79.02	2.000
600	0.63	3.001	7.52	2.000	0.63	3.001	78.97	2.000
Const.	0.63		7.5		0.63		79	

$$F1(k) = N^{\log_N(Flops) - k}$$

$$F2 = \log_N(Flops/Const)$$

Table 8.19: *FP and BTT floating point operations for network 31 as function of first layer size.*

	Flops						$\log_{N}(Flops)$				
	Gradient			Jacobian			Gradient		Jacobian		
S1	FP	BTT	BTT/FP	FP	BTT	BTT/FP	FP	BTT	FP	BTT	Weights (N)
1	4755	1955	0.411	6268	31886	5.087	5.26	4.71	5.43	6.44	5
2	13155	4135	0.314	14668	53066	3.618	3.96	3.47	4.00	4.54	11
3	29695	7495	0.252	31208	86066	2.758	3.50	3.03	3.51	3.86	19
4	60515	12455	0.206	62028	135106	2.178	3.27	2.80	3.28	3.51	29
5	111615	19255	0.173	113128	202706	1.792	3.13	2.66	3.13	3.29	41
6	189955	28135	0.148	191468	291386	1.522	3.03	2.56	3.04	3.14	55
10	948515	89255	0.094	950028	907306	0.955	2.82	2.34	2.82	2.81	131
20	10329315	505255	0.049	10330828	5157506	0.499	2.63	2.14	2.63	2.52	461
40	132158915	3281255	0.025	132160428	33841906	0.256	2.51	2.01	2.51	2.33	1721
60	617012515	10249255	0.017	617014028	106238306	0.172	2.46	1.96	2.46	2.24	3781
80	1870010115	23329255	0.012	1870011628	242506706	0.130	2.43	1.93	2.43	2.19	6641
100	4449871715	44441255	0.010	4449873228	462807106	0.104	2.40	1.91	2.40	2.16	10301
200	67559339715	337681255	0.005	67559341228	3530389106	0.052	2.35	1.85	2.35	2.07	40601

Table 8.20: *FP and BTT constant calculations for floating point operations for network 31 as a function of first layer size.*

	FP Gradient		BTT Gradient		FP Jacobian		BTT Jacobian		
S1	F1(2)	F2	F1(1.5)	F2	F1(2)	F2	F1(1.5)	F2	Weights (N)
1	190.20	2.969	174.86	2.401	250.72	3.140	2851.97	2.674	5
2	108.72	2.417	113.34	1.924	121.22	2.462	1454.55	2.007	11
3	82.26	2.245	90.50	1.769	86.45	2.262	1039.20	1.799	19
4	71.96	2.174	79.75	1.698	73.76	2.182	865.12	1.707	29
5	66.40	2.136	73.34	1.657	67.30	2.140	772.13	1.657	41
6	62.80	2.113	68.98	1.630	63.30	2.115	714.37	1.626	55
10	55.27	2.066	59.53	1.576	55.36	2.067	605.13	1.570	131
20	48.60	2.032	51.05	1.536	48.61	2.032	521.06	1.531	461
40	44.62	2.015	45.96	1.515	44.62	2.015	474.01	1.513	1721
60	43.16	2.009	44.08	1.509	43.16	2.009	456.95	1.507	3781
80	42.40	2.007	43.11	1.506	42.40	2.007	448.10	1.504	6641
100	41.94	2.005	42.51	1.504	41.94	2.005	442.67	1.503	10301
200	40.98	2.002	41.28	1.501	40.98	2.002	431.54	1.500	40601
Const.	40		41		40		431		

$$F1(k) = N^{\log_N(Flops) - k}$$

$$F2 = \log_N(Flops / Const)$$

Now we will describe experiments in which we change N by adjusting the number of delays in the network. Table 8.21 and Table 8.23 show the floating point operations required to compute the gradient and Jacobian for network 1 and network 31 as the number of delays is increased, for sequences of 20 samples and with the size of the first layer equal to 3 neurons. Table 8.22 and Table 8.24 use Eq. (8.4) and Eq. (8.5) to help determine the complexity of the algorithms.

The results here are simpler than those described on previous pages for changes in layer size. Here we find that the FP algorithms are $O(N^2)$ and the BTT algorithms are $O(N)$, and the result does not seem to be architecture dependent.

Table 8.21: *FP and BTT floating point operations for network 1 as function of number of delays.*

	Flops						$\log_N(Flops)$				
	Gradient			Jacobian			Gradient		Jacobian		
ND	FP	BTT	BTT/FP	FP	BTT	BTT/FP	FP	BTT	FP	BTT	Weights (N)
1	8802	3035	0.345	9923	34767	3.504	3.54	3.13	3.59	4.08	13
3	16402	3995	0.244	17523	44467	2.538	3.30	2.82	3.32	3.63	19
5	26882	4955	0.184	28003	54167	1.934	3.17	2.64	3.18	3.39	25
7	40152	5828	0.145	41363	63867	1.544	3.09	2.52	3.10	3.22	31
9	56392	6788	0.120	57603	73567	1.277	3.03	2.44	3.04	3.10	37
11	75512	7748	0.103	76723	83267	1.085	2.99	2.38	2.99	3.01	43
13	97512	8708	0.089	98723	92967	0.942	2.95	2.33	2.95	2.94	49
15	122392	9668	0.079	123603	102667	0.831	2.92	2.29	2.93	2.88	55
20	197192	12068	0.061	198403	126917	0.640	2.87	2.21	2.87	2.77	70
40	676394	21668	0.032	677603	223917	0.330	2.76	2.05	2.76	2.53	130
60	1443592	31268	0.022	1444803	320917	0.222	2.70	1.97	2.70	2.42	190
80	2498792	40868	0.016	2500003	417917	0.167	2.67	1.92	2.67	2.34	250
100	3841992	50468	0.013	3843203	514917	0.134	2.64	1.89	2.64	2.29	310
200	14878011	98468	0.007	14879203	999991	0.067	2.58	1.79	2.58	2.15	610

Table 8.22: *FP and BTT constant calculations for floating point operations for network 1 as a function of number of delays.*

	FP Gradient		BTT Gradient		FP Jacobian		BTT Jacobian		
ND	F1(2)	F2	F1(1)	F2	F1(2)	F2	F1(1)	F2	Weights (N)
1	52.08	2.103	233.46	1.145	58.72	2.150	2674.38	1.200	13
3	45.43	2.043	210.26	1.091	48.54	2.066	2340.37	1.129	19
5	43.01	2.023	198.20	1.065	44.80	2.035	2166.68	1.094	25
7	41.78	2.013	188.00	1.045	43.04	2.021	2060.23	1.074	31
9	41.19	2.008	183.46	1.036	42.08	2.014	1988.30	1.060	37
11	40.84	2.006	180.19	1.030	41.49	2.010	1936.44	1.051	43
13	40.61	2.004	177.71	1.025	41.12	2.007	1897.29	1.044	49
15	40.46	2.003	175.78	1.022	40.86	2.005	1866.67	1.038	55
20	40.24	2.001	172.40	1.016	40.49	2.003	1813.10	1.029	70
40	40.02	2.000	166.68	1.007	40.09	2.000	1722.44	1.015	130
60	39.99	2.000	164.57	1.004	40.02	2.000	1689.04	1.010	190
80	39.98	2.000	163.47	1.003	40.00	2.000	1671.67	1.008	250
100	39.98	2.000	162.80	1.002	39.99	2.000	1661.02	1.007	310
200	39.98	2.000	161.42	1.000	39.99	2.000	1639.33	1.004	610
Const.	40		161		40		1600		

$$F1(k) = N^{\log_N(Flops) - k}$$

$$F2 = \log_N(Flops/Const)$$

Table 8.23: *FP and BTT floating point operations for network 31 as function of number of delays.*

	Flops						$\log_N(Flops)$				
	Gradient			Jacobian			Gradient		Jacobian		
ND	FP	BTT	BTT/FP	FP	BTT	BTT/FP	FP	BTT	FP	BTT	Weights (N)
1	29695	7495	0.252	31208	86066	2.758	3.50	3.03	3.51	3.86	19
2	59315	11095	0.187	60828	123676	2.033	3.30	2.80	3.31	3.52	28
3	97575	14695	0.151	99088	161286	1.628	3.18	2.66	3.19	3.32	37
4	144475	18295	0.127	145988	198896	1.362	3.10	2.56	3.11	3.19	46
5	200015	21895	0.109	201528	236506	1.174	3.05	2.49	3.05	3.09	55
6	264195	25495	0.097	265708	274116	1.032	3.00	2.44	3.00	3.01	64
10	607315	39895	0.066	608828	424556	0.697	2.89	2.30	2.89	2.81	100
20	2069915	75895	0.037	2071428	800656	0.387	2.77	2.14	2.77	2.59	190
40	7587115	147895	0.019	7588628	1552856	0.205	2.68	2.01	2.68	2.41	370
60	16560317	219895	0.013	16561828	2305056	0.139	2.63	1.95	2.63	2.32	550
80	28989515	291895	0.010	28991031	3057256	0.105	2.61	1.91	2.61	2.26	730
100	44874715	363895	0.008	44876228	3809456	0.085	2.59	1.88	2.59	2.22	910

Table 8.24: *FP and BTT constant calculations for floating point operations for network 1 as a function of number of delays.*

	FP Gradient		BTT Gradient		FP Jacobian		BTT Jacobian		
ND	F1(2)	F2	F1(1)	F2	F1(2)	F2	F1(1)	F2	Weights (N)
1	82.26	2.143	394.47	0.995	86.45	2.160	4529.79	1.027	19
2	75.66	2.101	396.25	0.997	77.59	2.109	4417.00	1.016	28
3	71.27	2.077	397.16	0.998	72.38	2.081	4359.08	1.011	37
4	68.28	2.061	397.72	0.999	68.99	2.064	4323.83	1.008	46
5	66.12	2.051	398.09	0.999	66.62	2.052	4300.11	1.007	55
6	64.50	2.043	398.36	0.999	64.87	2.044	4283.06	1.006	64
10	60.73	2.026	398.95	0.999	60.88	2.026	4245.56	1.003	100
20	57.34	2.011	399.45	1.000	57.38	2.012	4213.98	1.001	190
40	55.42	2.004	399.72	1.000	55.43	2.004	4196.91	1.000	370
60	54.74	2.002	399.81	1.000	54.75	2.002	4191.01	1.000	550
80	54.40	2.001	399.86	1.000	54.40	2.001	4188.02	1.000	730
100	54.19	2.001	399.88	1.000	54.19	2.001	4186.22	1.000	910
Const.	54		400		54		4186		

$$F1(k) = N^{\log_N(Flops) - k}$$

$$F2 = \log_N(Flops/Const)$$

For the final experiments in this chapter we will investigate the effect of training sequence length, T, on the complexity of the algorithms. Table 8.25 and Table 8.27 presents the flops for network 1 and network 31 for a neural network with hidden layer size of 3 with 3 delays. The number of samples was increased from 20 to 5000. Each neural network has two layers, with 19 weights for network 1 and 37 weights for network 31. Table 8.26 and Table 8.28 use Eq. (8.4) and Eq. (8.5) to identify the complexity of the algorithms.

The main results from these tests are that the FP gradient and Jacobian algorithms and the BTT gradient algorithm are $O(T)$, while the BTT Jacobian algorithm is $O(T^2)$.

Table 8.25: *FP and BTT floating point operations for network 1 as a function of number of samples.*

	Flops						$\log_T(Flops)$			
	Gradient			Jacobian			Gradient		Jacobian	
T	FP	BTT	BTT/FP	FP	BTT	BTT/FP	FP	BTT	FP	BTT
20	16312	3908	0.240	17523	44467	2.538	3.24	2.76	3.26	3.57
40	32592	7768	0.238	35003	172067	4.916	2.82	2.43	2.84	3.27
60	48872	11628	0.238	52483	382867	7.295	2.64	2.29	2.65	3.14
80	65152	15488	0.238	69963	676867	9.675	2.53	2.20	2.55	3.06
100	81432	19348	0.238	84443	1054067	12.483	2.46	2.14	2.46	3.01
120	97712	23208	0.238	104923	1514467	14.434	2.40	2.10	2.41	2.97
140	113992	27068	0.237	122403	2058067	16.814	2.36	2.07	2.37	2.94
160	130272	30928	0.237	139883	2684867	19.194	2.32	2.04	2.33	2.92
180	146552	34788	0.237	157363	3394867	21.573	2.29	2.01	2.30	2.90
200	162832	38648	0.237	174843	4188067	23.953	2.26	1.99	2.28	2.88
220	179116	42508	0.237	192323	5064467	26.333	2.24	1.98	2.26	2.86
240	195396	46368	0.237	209803	6024069	28.713	2.22	1.96	2.24	2.85
260	212214	50768	0.239	227283	7066867	31.093	2.21	1.95	2.22	2.84
280	227952	54088	0.237	244763	(*)	(*)	2.19	1.93	2.20	(*)
300	244323	57948	0.237	262243	(*)	(*)	2.18	1.92	2.19	(*)
400	325632	77248	0.237	349643	(*)	(*)	2.12	1.88	2.13	(*)
500	407032	96548	0.237	437043	(*)	(*)	2.08	1.85	2.09	(*)
600	488432	115848	0.237	524443	(*)	(*)	2.05	1.82	2.06	(*)
1000	814032	193048	0.237	874043	(*)	(*)	1.97	1.76	1.98	(*)
5000	4070036	965048	0.237	4370043	(*)	(*)	1.79	1.62	1.80	(*)

(*) Tests were not performed due to computer memory limitations.

Table 8.26: *FP and BTT constant calculations for floating point operations for network 1 as a function of number of samples.*

	FP Gradient		BTT Gradient		FP Jacobian		BTT Jacobian	
T	F1(1)	F2	F1(1)	F2	F1(1)	F2	F1(2)	F2
20	815.60	1.0007	195.40	1.0041	876.15	1.0008	111.17	2.0222
40	814.80	1.0003	194.20	1.0017	875.08	1.0003	107.54	2.0091
60	814.53	1.0002	193.80	1.0010	874.72	1.0002	106.35	2.0055
80	814.40	1.0001	193.60	1.0007	874.54	1.0001	105.76	2.0038
100	814.32	1.0001	193.48	1.0005	844.43	0.9925	105.41	2.0029
120	814.27	1.0001	193.40	1.0004	874.36	1.0001	105.17	2.0023
140	814.23	1.0001	193.34	1.0004	874.31	1.0001	105.00	2.0019
160	814.20	1.0000	193.30	1.0003	874.27	1.0001	104.88	2.0017
180	814.18	1.0000	193.27	1.0003	874.24	1.0001	104.78	2.0014
200	814.16	1.0000	193.24	1.0002	874.22	1.0000	104.70	2.0013
220	814.16	1.0000	193.22	1.0002	874.20	1.0000	104.64	2.0011
240	814.15	1.0000	193.20	1.0002	874.18	1.0000	104.58	2.0010
260	816.21	1.0005	195.26	1.0021	874.17	1.0000	104.54	2.0009
280	814.11	1.0000	193.17	1.0002	874.15	1.0000	(*)	(*)
300	814.41	1.0001	193.16	1.0001	874.14	1.0000	(*)	(*)
400	814.08	1.0000	193.12	1.0001	874.11	1.0000	(*)	(*)
500	814.06	1.0000	193.10	1.0001	874.09	1.0000	(*)	(*)
600	814.05	1.0000	193.08	1.0001	874.07	1.0000	(*)	(*)
1000	814.03	1.0000	193.05	1.0000	874.04	1.0000	(*)	(*)
5000	814.01	1.0000	193.01	1.0000	874.01	1.0000	(*)	(*)
Const	814		193		874		104	

$$F1(k) = T^{\log_T(Flops) - k}$$

$$F2 = \log_T(Flops/Const)$$

(*) Tests were not performed due to computer memory limitations.

Table 8.27: *FP and BTT floating point operations for network 31 as a function of number of samples.*

	Flops						$\log_T(Flops)$			
	Gradient			Jacobian			Gradient		Jacobian	
T	FP	BTT	BTT/FP	FP	BTT	BTT/FP	FP	BTT	FP	BTT
20	97757	14695	0.150	99088	161286	1.628	3.84	3.20	3.84	4.00
40	195115	29335	0.150	198128	626896	3.164	3.30	2.79	3.31	3.62
60	292655	43975	0.150	297168	1396906	4.701	3.07	2.61	3.08	3.46
80	390195	58615	0.150	396208	2471316	6.237	2.94	2.51	2.94	3.36
100	487735	73255	0.150	495248	3850126	7.774	2.84	2.43	2.85	3.29
120	585275	87895	0.150	594288	5533336	9.311	2.77	2.38	2.78	3.24
140	682815	102535	0.150	693328	7520946	10.848	2.72	2.33	2.72	3.20
160	780355	117175	0.150	792368	9812956	12.384	2.67	2.30	2.68	3.17
180	877895	131815	0.150	891408	12409366	13.921	2.64	2.27	2.64	3.15
200	975435	146455	0.150	990448	15310176	15.458	2.60	2.24	2.61	3.12
220	1072975	161095	0.150	1089488	18515386	16.995	2.57	2.22	2.58	3.10
240	1170515	175735	0.150	1188528	22024996	18.531	2.55	2.20	2.55	3.08
260	1268145	190462	0.150	1287568	25839006	20.068	2.53	2.19	2.53	3.07
280	1365685	205102	0.150	1386608	(*)	(*)	2.51	2.17	2.51	(*)
300	1463225	219742	0.150	1485648	(*)	(*)	2.49	2.16	2.49	(*)
400	1950925	292942	0.150	1980848	(*)	(*)	2.42	2.10	2.42	(*)
500	2438625	366142	0.150	2476048	(*)	(*)	2.37	2.06	2.37	(*)
600	2928625	438142	0.150	2971348	(*)	(*)	2.33	2.03	2.33	(*)
1000	4877125	732142	0.150	4952048	(*)	(*)	2.23	1.95	2.23	(*)
5000	24385035	3660055	0.150	24790048	(*)	(*)	2.00	1.77	2.00	(*)

(*) Tests were not performed due to computer memory limitations.

Table 8.28: *FP and BTT constant calculations for floating point operations for network 31 as a function of number of samples.*

	FP Gradient		BTT Gradient		FP Jacobian		BTT Jacobian	
T	F1(1)	F2	F1(1)	F2	F1(1)	F2	F1(2)	F2
20	4887.85	1.0007	734.75	1.0013	4954.40	1.0002	403.22	2.0180
40	4877.88	1.0000	733.38	1.0005	4953.20	1.0001	391.81	2.0069
60	4877.58	1.0000	732.92	1.0003	4952.80	1.0000	388.03	2.0038
80	4877.44	1.0000	732.69	1.0002	4952.60	1.0000	386.14	2.0025
100	4877.35	1.0000	732.55	1.0002	4952.48	1.0000	385.01	2.0017
120	4877.29	1.0000	732.46	1.0001	4952.40	1.0000	384.26	2.0012
140	4877.25	1.0000	732.39	1.0001	4952.34	1.0000	383.72	2.0009
160	4877.22	1.0000	732.34	1.0001	4952.30	1.0000	383.32	2.0007
180	4877.19	1.0000	732.31	1.0001	4952.27	1.0000	383.01	2.0005
200	4877.18	1.0000	732.28	1.0001	4952.24	1.0000	382.75	2.0004
220	4877.16	1.0000	732.25	1.0001	4952.22	1.0000	382.55	2.0003
240	4877.15	1.0000	732.23	1.0001	4952.20	1.0000	382.38	2.0002
260	4877.48	1.0000	732.55	1.0001	4952.18	1.0000	382.23	2.0001
280	4877.45	1.0000	732.51	1.0001	4952.17	1.0000	(*)	(*)
300	4877.42	1.0000	732.47	1.0001	4952.16	1.0000	(*)	(*)
400	4877.31	1.0000	732.36	1.0001	4952.12	1.0000	(*)	(*)
500	4877.25	1.0000	732.28	1.0001	4952.10	1.0000	(*)	(*)
600	4881.04	1.0001	730.24	0.9996	4952.25	1.0000	(*)	(*)
1000	4877.13	1.0000	732.14	1.0000	4952.05	1.0000	(*)	(*)
5000	4877.01	1.0000	732.01	1.0000	4958.01	1.0001	(*)	(*)
Const	4877		732		4952		382	

$$F1(k) = T^{\log_T(Flops) - k}$$

$$F2 = \log_T(Flops/Const)$$

(*) Tests were not performed due to computer memory limitations.

8.4. Summary.

From the results presented in this chapter we confirmed the comments made at the end of Chapter 4 and Chapter 5, where we concluded that BTT is the best algorithm for gradient calculations and FP is the best algorithm for Jacobian calculation for some small networks and large sequences. The only case where the BTT Jacobian outperforms the FP Jacobian is for networks with a large number of weights and short training sequences. That combination rarely occurs. The BTT gradient requires about half of the time and fewer flops than the FP gradient. The BTT gradient required about 10 percent more memory than FP, a value not critical with systems available today. If we compare the Jacobian versions, FP requires about half of the memory, a third of the time and fewer flops than BTT. For very large sequences, the BTT and FP gradient tend to be close in computational complexity. It maybe best to use the FP gradient for long sequences, because of the memory requirements.

Another important point to be gained from this chapter is that algorithm complexity is dependent on the network architecture, and not just on the number of weights. Whether or not a network has feedback connections, and the number of feedback connections it has, can affect the algorithm complexity.

Chapter 9

SUMMARY

In this chapter a brief summary of the contributions of this research is presented, followed by recommendations for future work.

9.1. Summary of Results

There are two main contributions of this research. The first contribution is the development of two algorithms for gradient computation for a general class of dynamic network: the Layered Digital Dynamic Network. Chapter 4 presented the Forward Perturbation algorithm and Chapter 5 presented the Backpropagation-Through-Time algorithm. Each algorithm has a version for gradient-based optimization algorithms and Jacobian-based optimization algorithms. These two main algorithms have been introduced in the past by other authors, but this research has derived the complete equations for arbitrary LDDN's. We have shown that the LDDN structure can be used to represent most dynamic networks that have been proposed in the literature.

The second principal contribution of this research is the discovery of a key characteristic of the error surfaces of dynamic networks: spurious narrow valleys that can trap optimization algorithms. Chapter 6 analyzes these error surfaces and proposes new procedures that provide improved training for dynamic networks.

This research also demonstrated the application of the proposed algorithms to problems in control systems and nonlinear filtering (Chapter 7) and also compared the memory, the speed and the computational complexity characteristics of the FP and BTT algorithms (Chapter 8).

9.2. Future Work.

The work described in this dissertation has centered on exact gradient and Jacobian calculations for gradient-based and Jacobian-based optimization algorithms. Future research should explore faster approximate gradient and Jacobian calculations. It may be possible to obtain approximate gradients that are almost as affective as exact gradients for optimization algorithms, but that require significantly fewer computations.

REFERENCES

[1] Anderson, J.A., "A Simple Neural Network generating an Interactive Memory," *Mathematical Biosciences,* Vol. 14, 1972, pp. 197-220.

[2] Atiya, A.F.; Parlos, A.G., "New Results on Recurrent Network Training: Unifying the Algorithms and Accelerating Convergence," *IEEE Transactions on Neural networks,* Vol. 11, 2000, pp. 697-709.

[3] Back, A.D.; Tsoi, A.C., "FIR and IIR Synapses, a new Neural Network Architecture for Time Series Modelling," *Neural Computation,* Vol. 3, 1991, pp. 375-385.

[4] Box, G.E.P.; Jenkins, G.M.; Reinsel, G.C., *Time series analysis : forecasting and control,* Englewood Cliffs, N.J.: Prentice Hall, 1994.

[5] Brengel, D.D.; Seider, W.D., "Multistep Nonlinear Predictive Controller," *Ind. Eng. Chem. Res.,* Vol. 28, 1989, pp. 1812-1822.

[6] Chen, D.; Hagan, M.T., "Optimal use of regularization and cross-validation in neural network modeling," *Proceedings. IJCNN '99. International Joint Conference on Neural Networks,* Washington, DC, July, 1999, Vol. 2, pp. 1275 - 1280.

[7] De Jesús, O., "Reinforcement Learning Control Applied to Diesel Engines," *Master Thesis,* Oklahoma State University, Stillwater, 1998.

[8] De Jesús, O.; Hagan, M.T., "Backpropagation through time for a general class of dynamic network," *Proceedings. IJCNN '01. International Joint Conference on Neural Networks,* Washington, DC, July, 2001, Vol. 4, pp. 2638 - 2643.

[9] De Jesús, O.; Hagan, M.T., "Forward perturbation for a general class of dynamic network," *Proceedings. IJCNN '01. International Joint Conference on Neural Networks,* Washington, DC, July, 2001, Vol. 4, pp. 2626 - 2631.

[10] De Jesús, O.; Horn, J.M.; Hagan, M.T., "Analysis of Recurrent Network Training and Suggestions for Improvements," *Proceedings. IJCNN '01. International Joint Conference on Neural Networks,* Washington DC, July 2001, Vol. 4, pp. 2632 - 2637.

[11] De Jesús, O.; Pukrittayakamee, A.; Hagan, M.T., "A Comparison of Neural Network Control Algorithms," *Proceedings. IJCNN '01. International Joint Conference on Neural Networks,* Washington DC, July 2001, Vol. 1, pp. 521-526.

[12] Demuth, H. B. and Beale, M., *Users' Guide for the Neural Network Toolbox for MATLAB,* Natick, MA: The Mathworks, 1998.

[13] De Vries, B.; Principe, J.C., "The Gamma Model - A New Neural Model for Temporal Processing," *Neural Networks,* Vol. 5, 1992, pp. 565-576.

[14] Elman, J., "Finding Structure in Time," *Cognitive Science,* Vol. 14, 1990, pp. 179-211.

[15] Feldkamp, L.A.; Prokhorov, D.V., "Phased Backpropagation: A Hybrid of BPTT and Temporal BP," Neural Networks Proceedings, 1998. IEEE World Congress on Computational Intelligence. Vol. 3, may 1998, pp. 2262 - 2267.

[16] Foresee, D.F.; Hagan, M.T., "Gauss-Newton approximation to Bayesian learning," *Proceedings. IJCNN '97. International Joint Conference on Neural Networks,* Houston, TX, June, 1997, Vol. 3, pp. 1930 - 1935.

[17] Frasconi, P.; Gori, M.; Soda, G., "Local Feedback Multilayered Networks," Neural Computation, Vol. 4, 1992, pp. 120-130.

[18] Grossberg, A., "How does the Brain build a Cognitive Code ?," Psychological Review, Vol. 88, 1980, pp. 375-407.

[19] Hagan, M.T.; De Jesús, O.; Schultz, R.L., "Training Recurrent Networks for Filtering and Control," *In Recurrent Neural Networks: Design and Applications,* L.R. Medsker and L.C. Jain, Eds., CRC Press, 2000, pp. 325-354.

[20] Hagan, M.T.; Demuth, H. B.; Beale, M., *Neural Network Design,* Boston, MA: PWS Publishing Company, 1996.

[21] Hagan, M.T.; Menhaj, M., "Training Feedforward Networks with the Marquardt Algorithm," *IEEE Transactions on Neural Networks,* Vol. 5, 1994, pp. 989-993.

[22] Hebb, D.O., *The Organization of Behavior,* New York, NY: Wiley, 1949.

[23] Hopfield, J.J., "Neural Networks and Physical Systems with emergent collective Computational abilities," *Proceedings of the National Academy of Sciences,* Vol. 79, 1982, pp. 2554-2558.

[24] Hsu, H.H.; Fu, L.M.; Principe, J.C., "Context Analysis by the Gamma Neural Network," *IEEE International Conference on Neural Networks,* Vol. 2, 1996, pp. 682-687.

[25] Kohonen, T., "Correlation Matrix Memories," *IEEE Transactions on Computers,* Vol. 21, 1972, pp. 353-359.

[26] Lapedes, A.S; Farber R., *Nonlinear Signal Processing using Neural Networks: Prediction and System Modelling,* Los Alamos, NM: Los Alamos National Laboratory LA-UR-87-2662, 1987.

[27] Li, L.; Haykin, S., "A cascaded recurrent neural network for real-time nonlinear adaptive filtering," *IEEE International Conference on Neural Networks,* Vol. 2, 1993, pp. 857-862.

[28] Mackey, M.C.; Glass, L., "Oscillation and Chaos in Physiological Control Systems," *Science,* Vol. 197, 1977, pp. 287-289.

[29] Magnus, J.R.; Neudecker, H., *Matrix Differential Calculus,* Chichester, UK: John Wiley & Sons, Ltd., 1999.

[30] McCulloch, W.; Pitts, W., "A Logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics,* Vol. 5, 1943, pp. 115-133.

[31] Mills, P.M.;Zomaya, A.Y.; Tade, M.O., *Neuron-Adaptive Process Control: a Practical Approach,* New York, NY: John Wiley & Sons, 1995.

[32] Minsky, M.; Papert, S., *Perceptrons,* Cambridge, MA: MIT Press, 1969.

[33] Motter, M.A.; Principe, J.C., "A Gamma Memory Neural network for System Identification," *IEEE International Conference on Neural Networks,* Vol. 5, 1994, pp. 3232-3237.

[34] Narendra, K.S.; Mukhopadhyay, S., "Adaptive Control using Neural Networks and Approximate Models," *IEEE Transactions on Neural Networks,* Vol. 8, 1997, pp. 475-485.

[35] Narendra, K.S.; Parthasrathy, A.M., "Identification and control for dynamic systems using neural networks," *IEEE Transactions on Neural Networks,* Vol. 1, 1990, pp. 4-27.

[36] Nerrand, O.; Roussel-Ragot, P.; Personnaz, L.; Dreyfus, G.; Marcos, S., "Neural Networks and Nonlinear Adaptive Filtering: Unifying Concepts and New Algorithms," *Neural Computation,* Vol. 5, 1993, pp. 165-197.

[37] Oppenheim, A.V.; Schafer, R.W., *Discrete-Time Signal Processing,* Englewood Cliffs, NJ: Prentice Hall, 1989.

[38] Poggio, T.; Girosi, F., "Networks for Approximation and Learning," *Proceedings of the IEEE,* Vol.78, 1990, pp. 1481-1497.

[39] Principe, J.C.; De Vries, B.; De Oliveira, P.G., "The Gamma Filter - A New Class of Adaptive IIR Filters with Restricted Feedback," *IEEE Transactions on Signal Processing,* Vol. 41, 1993, pp. 649-656.

[40] Principe, J.C.; Euliano, N.R.; Lefebvre, W.C., *Neural and Adaptive Systems,* New York, NY: John Wiley & Sons, 2000.

[41] Rosenblatt, F., "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review,* Vol. 65, 1958, pp. 386-408.

[42] Rumelhart, D.; Hinton, D.; Williams, G., "Learning Internal Representations by Error Propagation," *In: Parallel Distributed Processing,* Vol. 1, Rumelhart, D; McClelland, F., Eds., Cambridge, MA: M.I.T. Press, 1986.

[43] Scales, L.E., *Introduction to non-linear optimization,* New York, NY: Springer-Verlag, 1985.

[44] Schmidhuber, J., "A Fixed Size Storage O(n^3) Time Complexity Learning Algorithm for Fully Recurrent Continually Running Networks," *Neural Computation,* Vol. 4, 1992, pp. 243 - 248.

[45] Sun, G.Z.; Chen, H.H.; Lee, Y.C., "Green's Method for Fast On-Line Learning Algorithm of Recurrent Neural Networks," *In: Advances in Neural Information Processing Systems 4,* Moody J.; Hanson S.; Lippmann R. (eds), San Mateo, CA: Morgan Kaufmann, 1992, pp. 333 - 340.

[46] Toomarian, N.; Barben, J., "Adjoint-Functions and Temporal Learning Algorithms in Neural Networks," *In: Advances in Neural Information Processing Systems 3,* Lippmann R.; Moody J., Touretzky D. (eds), San Mateo, CA: Morgan Kaufmann, 1991, pp. 113 - 120.

[47] Toomarian, N.; Barben, J., "Learning a Trajectory Using Adjoint Functions and Teacher Forcing," *Neural Networks,* Vol. 5, 1992, pp. 473 - 484.

[48] Tsoi, A.C., "Architectures and Learning in Recurrent Neural Networks," *In: Adaptive Processing and Data Structures,* Giles C.L.; Gori M. (Eds.), Berlin, Germany: Springer, 1998, pp. 1 - 26.

[49] Waibel, A.; Hanazawa, T.; Hinton, G.; Shikano, K.; Lang, K.J., "Phoneme recognition using time-delay neural networks," *IEEE Transactions on Acoustics, Speech and Signal Processing,* Vol. 37, 1989, pp. 328 -339.

[50] Wan, E.A., "Temporal backpropagation for FIR neural networks," *1990 IJCNN International Joint Conference on Neural Networks,* Vol. 1, 1990, pp. 575 - 580.

[51] Werbos, P. J., "Backpropagation through time: What it is and how to do it," *Proceedings of the IEEE,* Vol. 78, 1990, pp. 1550–1560.

[52] Widrow, B.; Hoff, M.E., "Adaptive Switching Circuits," *1960 IRE WESCON Convention Record,* New York, NY, IRE Part 4, 1960, pp. 96-104.

[53] Williams, R.J.; Peng, J., "Gradient-based Learning Algorithms for Recurrent Neural Networks and their Computational Complexity," *In: Backpropagation: Theory, Architectures, and Applications,* Chauvin, Y.; Rumelhart, D.E. (Eds.), Hillsdale, NJ: Lawrence Erlbaum, 1992, pp. 433-486.

[54] Williams, R.J.; Zipser, D., "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation,* Vol. 1, pp. 270–280, 1989.

[55] Yang, W., *Neurocontrol Using Dynamic Learning,* Doctoral Thesis, Oklahoma State University, 1994.

[56] Yang, W.; Hagan, M.T., "Training recurrent networks," *Proceedings of the 7th Oklahoma Symposium on Artificial Intelligence,* Stillwater, OK, 1993, pp. 226 - 233.

Appendix A

NEURAL NETWORK EXAMPLES USING THE LAYERED DIGITAL DYNAMIC NETWORK (LDDN)

This appendix illustrates how we can implement some neural network architectures explained in section 2.2 using the LDDN. For simplicity we will describe the neural network examples with the minimum number of layers.

A.1. Narendra Models.

For the Narendra Models the input *u(k)* will be represented by $p^1(t)$ and the output $y_p(k)$ will be represented by the output of the last layer $\mathbf{a}^M(t)$, where M is the last layer in the network.

The Model I (Eq. (2.1)) could be represented as:

$$\mathbf{a}^1(t) = \mathbf{f}^1\left(\mathbf{IW}^{1,1}\begin{bmatrix} p^1(t-1) \\ p^1(t-2) \\ \dots \\ p^1(t-m) \end{bmatrix} + \mathbf{b}^1\right)$$

$$\mathbf{a}^2(t) = \mathbf{f}^2\left(\mathbf{LW}^{2,2}\begin{bmatrix} \mathbf{a}^2(t-1) \\ \mathbf{a}^2(t-2) \\ \dots \\ \mathbf{a}^2(t-n) \end{bmatrix} + \mathbf{LW}^{2,1}\mathbf{a}^1(t) + \mathbf{b}^2\right) = \mathbf{LW}^{2,2}\begin{bmatrix} \mathbf{a}^2(t-1) \\ \mathbf{a}^2(t-2) \\ \dots \\ \mathbf{a}^2(t-n) \end{bmatrix} + \mathbf{a}^1(t) \qquad \textbf{(A.1)}$$

The Model II (Eq. (2.2)) could be represented as:

$$\mathbf{a}^1(t) = \mathbf{f}^1\left(\mathbf{LW}^{1,2}\begin{bmatrix}\mathbf{a}^2(t-1)\\ \mathbf{a}^2(t-2)\\ \dots\\ \mathbf{a}^2(t-n)\end{bmatrix} + \mathbf{b}^1\right)$$

$$\mathbf{a}^2(t) = \mathbf{f}^2\left(\mathbf{IW}^{2,1}\begin{bmatrix}p^1(t-1)\\ p^1(t-2)\\ \dots\\ p^1(t-m)\end{bmatrix} + \mathbf{LW}^{2,1}\mathbf{a}^1(t) + \mathbf{b}^2\right) = \mathbf{IW}^{2,1}\begin{bmatrix}p^1(t-1)\\ p^1(t-2)\\ \dots\\ p^1(t-m)\end{bmatrix} + \mathbf{a}^1(t) \qquad \textbf{(A.2)}$$

The Model III (Eq. (2.3)) could be represented as:

$$\mathbf{a}^1(t) = \mathbf{f}^1\left(\mathbf{IW}^{1,1}\begin{bmatrix}p^1(t-1)\\ p^1(t-2)\\ \dots\\ p^1(t-m)\end{bmatrix} + \mathbf{b}^1\right)$$

$$\mathbf{a}^2(t) = \mathbf{f}^2\left(\mathbf{LW}^{2,2}\begin{bmatrix}\mathbf{a}^2(t-1)\\ \mathbf{a}^2(t-2)\\ \dots\\ \mathbf{a}^2(t-n)\end{bmatrix} + \mathbf{b}^2\right) \qquad \textbf{(A.3)}$$

$$\mathbf{a}^3(t) = \mathbf{f}^3(\mathbf{LW}^{3,1}\mathbf{a}^1(t) + \mathbf{LW}^{3,2}\mathbf{a}^2(t) + \mathbf{b}^3) = \mathbf{a}^1(t) + \mathbf{a}^2(t)$$

The Model IV (Eq. (2.4)) could be represented as:

$$\mathbf{a}^1(t) = \mathbf{f}^1\left(\mathbf{IW}^{1,1}\begin{bmatrix}p^1(t-1)\\ p^1(t-2)\\ \dots\\ p^1(t-m)\end{bmatrix} + \mathbf{LW}^{1,1}\begin{bmatrix}\mathbf{a}^1(t-1)\\ \mathbf{a}^1(t-2)\\ \dots\\ \mathbf{a}^1(t-n)\end{bmatrix} + \mathbf{b}^1\right) \qquad \textbf{(A.4)}$$

For the previous four equations the last layer is the output layer. The transfer function for the last layer must be linear for Eq. (2.1), Eq. (2.2) and Eq. (2.3). From the examples, Model IV requires fewer layers. All the other examples required an additional layer with a linear transfer function.

For the Narma-L1 Model (Eq. (2.5)) with *m=2* we have:

$$
\begin{aligned}
\mathbf{a}^1(t) &= \mathbf{f}^1\left(\mathbf{LW}^{1,6}\begin{bmatrix}\mathbf{a}^6(t-1)\\ \mathbf{a}^6(t-2)\\ \cdots\\ \mathbf{a}^6(t-n)\end{bmatrix}+\mathbf{b}^1\right)\\
\mathbf{a}^2(t) &= \mathbf{f}^2\left(\mathbf{LW}^{2,6}\begin{bmatrix}\mathbf{a}^6(t-1)\\ \mathbf{a}^6(t-2)\\ \cdots\\ \mathbf{a}^6(t-n)\end{bmatrix}+\mathbf{b}^2\right)\\
\mathbf{a}^3(t) &= \mathbf{f}^3(\mathbf{IW}^{3,1}p^1(t-1)\cdot\mathbf{LW}^{3,2}\mathbf{a}^2(t)\cdot\mathbf{b}^3) = p^1(t-1)\cdot\mathbf{a}^2(t)\\
\mathbf{a}^4(t) &= \mathbf{f}^4\left(\mathbf{LW}^{4,6}\begin{bmatrix}\mathbf{a}^6(t-1)\\ \mathbf{a}^6(t-2)\\ \cdots\\ \mathbf{a}^6(t-n)\end{bmatrix}+\mathbf{b}^4\right)\\
\mathbf{a}^5(t) &= \mathbf{f}^5(\mathbf{IW}^{5,1}p^1(t-2)\cdot\mathbf{LW}^{5,4}\mathbf{a}^4(t)\cdot\mathbf{b}^5) = p^1(t-2)\cdot\mathbf{a}^4(t)\\
\mathbf{a}^6(t) &= \mathbf{f}^6(\mathbf{LW}^{6,1}\mathbf{a}^1(t)+\mathbf{LW}^{6,3}\mathbf{a}^3(t)+\mathbf{LW}^{6,5}\mathbf{a}^5(t)+\mathbf{b}^6) = \mathbf{a}^1(t)+\mathbf{a}^3(t)+\mathbf{a}^5(t)
\end{aligned}
\tag{A.5}
$$

where the transfer functions for layers 3, 5 and 6 must be linear. The first layer represents the *f* network, each subsequent pair represents the networks g_0 and g_1, where we have a nonlinear layer followed by a linear layer with a product operation instead of a summation. We set the biases of layers 3 and 5 to one. The last layer combines the result of the previous three subnetworks.

The Narma-L2 Model (Eq. (2.6)) will be represented by:

$$
\begin{aligned}
\mathbf{a}^1(t) &= \mathbf{f}^1\left(\mathbf{IW}^{1,1}\begin{bmatrix}p^1(t-2)\\ p^1(t-3)\\ \cdots\\ p^1(t-n)\end{bmatrix}+\mathbf{LW}^{1,4}\begin{bmatrix}\mathbf{a}^4(t-1)\\ \mathbf{a}^4(t-2)\\ \cdots\\ \mathbf{a}^4(t-n)\end{bmatrix}+\mathbf{b}^1\right)\\
\mathbf{a}^2(t) &= \mathbf{f}^2\left(\mathbf{IW}^{2,1}\begin{bmatrix}p^1(t-2)\\ p^1(t-3)\\ \cdots\\ p^1(t-n)\end{bmatrix}+\mathbf{LW}^{1,4}\begin{bmatrix}\mathbf{a}^4(t-1)\\ \mathbf{a}^4(t-2)\\ \cdots\\ \mathbf{a}^4(t-n)\end{bmatrix}+\mathbf{b}^2\right)\\
\mathbf{a}^3(t) &= \mathbf{f}^3(\mathbf{IW}^{3,1}p^1(t-1)\cdot\mathbf{LW}^{3,2}\mathbf{a}^2(t)\cdot\mathbf{b}^3) = p^1(t-1)\cdot\mathbf{a}^2(t)\\
\mathbf{a}^4(t) &= \mathbf{f}^4(\mathbf{LW}^{4,1}\mathbf{a}^1(t)+\mathbf{LW}^{4,3}\mathbf{a}^3(t)+\mathbf{b}^4) = \mathbf{a}^1(t)+\mathbf{a}^3(t)
\end{aligned}
\tag{A.6}
$$

As in the Narma-L1 model, Layer 3 allow us to multiply the last input to the model times the output of the subnetwork g_0. We could generate the neurocontroller shown in Eq. (2.7) by rearranging Eq. (2.6):

$$
\begin{aligned}
\mathbf{a}^1(t) &= \mathbf{f}^1\left(\mathbf{IW}^{1,1}\begin{bmatrix} p^1(t-2) \\ p^1(t-3) \\ \cdots \\ p^1(t-n) \end{bmatrix} + \mathbf{LW}^{1,4}\begin{bmatrix} \mathbf{a}^4(t-1) \\ \mathbf{a}^4(t-2) \\ \cdots \\ \mathbf{a}^4(t-n) \end{bmatrix} + \mathbf{b}^1\right) \\
\mathbf{a}^2(t) &= \mathbf{f}^2\left(\mathbf{IW}^{2,1}\begin{bmatrix} p^1(t-2) \\ p^1(t-3) \\ \cdots \\ p^1(t-n) \end{bmatrix} + \mathbf{LW}^{1,4}\begin{bmatrix} \mathbf{a}^4(t-1) \\ \mathbf{a}^4(t-2) \\ \cdots. \\ \mathbf{a}^4(t-n) \end{bmatrix} + \mathbf{b}^2\right) \\
\mathbf{a}^3(t) &= \mathbf{f}^3(\mathbf{IW}^{3,2}p^2(t) - \mathbf{LW}^{3,1}\mathbf{a}^1(t) + \mathbf{b}^3) = p^2(t) - \mathbf{a}^1(t) \\
\mathbf{a}^4(t) &= \mathbf{f}^4((\mathbf{LW}^{4,3}\mathbf{a}^3(t))/(\mathbf{LW}^{4,3}\mathbf{a}^2(t))/(\mathbf{b}^4)) = \mathbf{a}^3(t)/\mathbf{a}^2(t)
\end{aligned}
\tag{A.7}
$$

where $p^2(t)$ is the desired output. For the last two models we included product and division operators inside the evaluation of the transfer function input. Those operators are considered in the final implementations of the dynamic gradient calculation algorithms.

A.2. Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) layers.

A FIR layer is implemented as the product of a weight times the input to the layer, where that input could be external or the output from another layer:

$$
\mathbf{a}^M(t) = \mathbf{f}^M\left(\mathbf{IW}^{M,N}\begin{bmatrix} p^N(t-1) \\ p^N(t-2) \\ \cdots \\ p^N(t-n) \end{bmatrix}\right) \text{ or } \mathbf{a}^M(t) = \mathbf{f}^M\left(\mathbf{LW}^{M,N}\begin{bmatrix} a^N(t-1) \\ a^N(t-2) \\ \cdots \\ a^N(t-n) \end{bmatrix}\right) \tag{A.8}
$$

Similarly, an IIR layer is implemented as the product of a weight times the input to the layer plus a delayed feedback of the same layer, where the input to the filter could be external or the output from another layer:

$$
\mathbf{a}^M(t) = \mathbf{f}^M\left(\mathbf{IW}^{M,N}\begin{bmatrix} p^N(t-1) \\ p^N(t-2) \\ \cdots \\ p^N(t-n) \end{bmatrix} + \mathbf{LW}^{M,M}\begin{bmatrix} a^M(t-1) \\ a^M(t-2) \\ \cdots \\ a^M(t-m) \end{bmatrix}\right) \text{ or} \tag{A.9}
$$

$$
\mathbf{a}^M(t) = \mathbf{f}^M\left(\mathbf{LW}^{M,N}\begin{bmatrix} a^N(t-1) \\ a^N(t-2) \\ \cdots \\ a^N(t-n) \end{bmatrix} + \mathbf{LW}^{M,M}\begin{bmatrix} a^M(t-1) \\ a^M(t-2) \\ \cdots \\ a^M(t-m) \end{bmatrix}\right) \tag{A.10}
$$

A.3. Gamma Memory Structure.

We can construct the Gamma Memory Structure using the following configuration with a one-layer network:

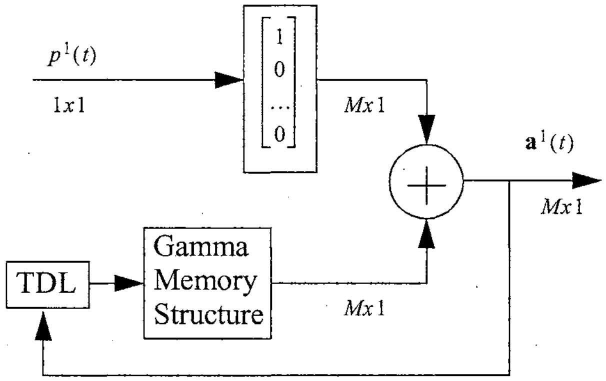

Figure A.1: *Gamma Memory Structure.*

We can create the filter with a one-layer NN:

$$\begin{aligned}\mathbf{a}^1(t) &= \mathbf{IW}^{11}p^1(t) + GM(w, a^1(t-1), \dots, a^1(t-M)) \\ &= \begin{bmatrix}1\\0\\\dots\\0\end{bmatrix}p^1(t) + \begin{bmatrix}0 & 0 & 0 & \dots & 0\\ w & (1-w) & 0 & \dots & 0\\ 0 & w & (1-w) & \dots & 0\\ \dots & \dots & \dots & \dots & \dots\\ 0 & 0 & w & \dots & (1-w)\end{bmatrix}\begin{bmatrix}a^1(t-1)\\a^1(t-2)\\\dots\\a^1(t-M)\end{bmatrix}\end{aligned} \quad \textbf{(A.11)}$$

where the input weight $\mathbf{IW}^{11}$ is not being trained.

Appendix B

COMPARISON OF CALCULATION PROCEDURES FOR DIFFERENT DYNAMIC GRADIENT CALCULATION METHODS

This chapter will show an example based on the simple neural network presented in Figure 3.1 on page 20 and Figure 6.2 on page 53. For a sequence of four points we will demonstrate Forward Perturbation (Eq. (3.51) and Eq. (3.53)), Backpropagation-Through-Time (Eq. (3.52) and Eq. (3.54)), the Green's Function Method (Eq. (3.60) to Eq. (3.62)), the Fast Forward Propagation Method (Eq. (3.65) to Eq. (3.69)) and the Block Update method (Eq. (3.70) and Eq. (3.84)).

If we apply Eq. (6.1) from $t = 1$ to $t = 4$, we will obtain:

$$y(1) = \mathbf{W}^1 u(1) + \mathbf{W}^2 y(0) \tag{B.1}$$

$$y(2) = \mathbf{W}^1 u(2) + \mathbf{W}^2 y(1) \tag{B.2}$$

$$y(3) = \mathbf{W}^1 u(3) + \mathbf{W}^2 y(2) \tag{B.3}$$

$$y(4) = \mathbf{W}^1 u(4) + \mathbf{W}^2 y(3) \tag{B.4}$$

The dynamic Forward Perturbation equations (Eq. (3.53)) for $\mathbf{W}^1$ are:

$$\frac{\partial y(1)}{\partial \mathbf{W}^1} = \frac{\partial^e y(1)}{\partial \mathbf{W}^1} + \frac{\partial^e y(1)}{\partial y(0)} \times \frac{\partial y(0)}{\partial \mathbf{W}^1} = u(1) \tag{B.5}$$

$$\frac{\partial y(2)}{\partial \mathbf{W}^1} = \frac{\partial^e y(2)}{\partial \mathbf{W}^1} + \frac{\partial^e y(2)}{\partial y(1)} \times \frac{\partial y(1)}{\partial \mathbf{W}^1} = u(2) + \mathbf{W}^2 u(1) \tag{B.6}$$

$$\frac{\partial y(3)}{\partial \mathbf{W}^1} = \frac{\partial^e y(3)}{\partial \mathbf{W}^1} + \frac{\partial^e y(3)}{\partial y(2)} \times \frac{\partial y(2)}{\partial \mathbf{W}^1} = u(3) + \mathbf{W}^2 u(2) + (\mathbf{W}^2)^2 u(1) \tag{B.7}$$

$$\frac{\partial y(4)}{\partial \mathbf{W}^1} = \frac{\partial^e y(4)}{\partial \mathbf{W}^1} + \frac{\partial^e y(4)}{\partial y(3)} \times \frac{\partial y(3)}{\partial \mathbf{W}^1} = u(4) + \mathbf{W}^2 u(3) + (\mathbf{W}^2)^2 u(2) + (\mathbf{W}^2)^3 u(1) \tag{B.8}$$

We will obtain the final gradient for $\mathbf{W}^1$ by applying Eq. (3.51):

$$\begin{aligned}\frac{\partial F}{\partial \mathbf{W}^1} &= \sum_{t=1}^{Q}\left[\frac{\partial y(t)}{\partial \mathbf{W}^1}\right]^T \times \frac{\partial^e F}{\partial y(t)} \\ &= -u(1)(t(1)-y(1)) - (u(2)+\mathbf{W}^2 u(1))(t(2)-y(2)) \\ &\quad -(u(3)+\mathbf{W}^2 u(2)+(\mathbf{W}^2)^2 u(1))(t(3)-y(3)) \\ &\quad -(u(4)+\mathbf{W}^2 u(3)+(\mathbf{W}^2)^2 u(2)+(\mathbf{W}^2)^3 u(1))(t(4)-y(4))\end{aligned} \tag{B.9}$$

To obtain the gradient of $\mathbf{W}^1$ using FP we required 7 multiplications and 10 additions.

The dynamic Backpropagation-Through-Time equations (Eq. (3.54)) for $\mathbf{W}^1$ are:

$$\frac{\partial F}{\partial y(4)} = \frac{\partial^e F}{\partial y(4)} + \frac{\partial^e y(5)}{\partial y(4)} \times \frac{\partial F}{\partial y(5)} = -(t(4)-y(4)) \tag{B.10}$$

$$\frac{\partial F}{\partial y(3)} = \frac{\partial^e F}{\partial y(3)} + \frac{\partial^e y(4)}{\partial y(3)} \times \frac{\partial F}{\partial y(4)} = -(t(3)-y(3)) - \mathbf{W}^2(t(4)-y(4)) \tag{B.11}$$

$$\frac{\partial F}{\partial y(2)} = \frac{\partial^e F}{\partial y(2)} + \frac{\partial^e y(3)}{\partial y(2)} \times \frac{\partial F}{\partial y(3)} = -(t(2)-y(2)) - \mathbf{W}^2(t(3)-y(3)) - (\mathbf{W}^2)^2(t(4)-y(4)) \tag{B.12}$$

$$\begin{aligned}\frac{\partial F}{\partial y(1)} = \frac{\partial^e F}{\partial y(1)} + \frac{\partial^e y(2)}{\partial y(1)} \times \frac{\partial F}{\partial y(1)} = &-(t(1)-y(1)) - \mathbf{W}^2(t(2)-y(2)) - (\mathbf{W}^2)^2(t(3)-y(3)) \\ &-(\mathbf{W}^2)^3(t(4)-y(4))\end{aligned} \tag{B.13}$$

We will obtain the final gradient for $\mathbf{W}^1$ by applying Eq. (3.52):

$$\begin{aligned}\frac{\partial F}{\partial \mathbf{W}^1} &= \sum_{t=1}^{Q}\left[\frac{\partial^e y(t)}{\partial \mathbf{W}^1}\right]^T \times \frac{\partial F}{\partial y(t)} \\ &= -u(1)((t(1)-y(1)) + \mathbf{W}^2(t(2)-y(2)) + (\mathbf{W}^2)^2(t(3)-y(3)) + (\mathbf{W}^2)^3(t(4)-y(4))) \\ &\quad -u(2)((t(2)-y(2)) + \mathbf{W}^2(t(3)-y(3)) + (\mathbf{W}^2)^2(t(4)-y(4))) \\ &\quad -u(3)((t(3)-y(3)) + \mathbf{W}^2(t(4)-y(4))) - u(4)(t(4)-y(4))\end{aligned} \tag{B.14}$$

where this results is the same for the FP in Eq. (B.9). To obtain the gradient of $\mathbf{W}^1$ using BTT we required 7 multiplications and 6 additions.

The Green's Function Method must be applied recursively by using Eq. (3.60) to Eq. (3.62):

$$U(1) = 1 \tag{B.15}$$

$$S(1) = (U(1))^{-1} \times \frac{\partial^e y(1)}{\partial \mathbf{W}^1} + S(0) = u(1) \tag{B.16}$$

$$\left.\frac{\partial F}{\partial \mathbf{W}^1}\right|_1 = \left.\frac{\partial F}{\partial \mathbf{W}^1}\right|_0 + [U(1) \times S(1)]^T \times \frac{\partial^e F}{\partial y(1)} = -u(1)(t(1)-y(1)) \tag{B.17}$$

$$U(2) = \frac{\partial^e \mathbf{a}(2)}{\partial \mathbf{a}(1)} \times U(1) = \mathbf{W}^2 \tag{B.18}$$

$$S(2) = (U(2))^{-1} \times \frac{\partial^e y(2)}{\partial \mathbf{W}^1} + S(1) = \frac{u(2)}{\mathbf{W}^2} + u(1) \tag{B.19}$$

$$\begin{aligned} \left.\frac{\partial F}{\partial \mathbf{W}^1}\right|_2 &= \left.\frac{\partial F}{\partial \mathbf{W}^1}\right|_1 + [U(2) \times S(2)]^T \times \frac{\partial^e F}{\partial y(2)} \\ &= -u(1)(t(1)-y(1)) - (u(2) + \mathbf{W}^2 u(1))(t(2)-y(2)) \end{aligned} \tag{B.20}$$

$$U(3) = \frac{\partial^e \mathbf{a}(3)}{\partial \mathbf{a}(2)} \times U(2) = (\mathbf{W}^2)^2 \tag{B.21}$$

$$S(3) = (U(3))^{-1} \times \frac{\partial^e y(3)}{\partial \mathbf{W}^1} + S(2) = \frac{u(3)}{(\mathbf{W}^2)^2} + \frac{u(2)}{\mathbf{W}^2} + u(1) \tag{B.22}$$

$$\begin{aligned} \left.\frac{\partial F}{\partial \mathbf{W}^1}\right|_3 &= \left.\frac{\partial F}{\partial \mathbf{W}^1}\right|_2 + [U(3) \times S(3)]^T \times \frac{\partial^e F}{\partial y(3)} \\ &= -u(1)(t(1)-y(1)) - (u(2) + \mathbf{W}^2 u(1))(t(2)-y(2)) \\ &\quad -(u(3) + \mathbf{W}^2 u(2) + (\mathbf{W}^2)^2 u(1))(t(3)-y(3)) \end{aligned} \tag{B.23}$$

$$U(4) = \frac{\partial^e \mathbf{a}(4)}{\partial \mathbf{a}(3)} \times U(3) = (\mathbf{W}^2)^3 \tag{B.24}$$

$$S(3) = (U(4))^{-1} \times \frac{\partial^e y(4)}{\partial \mathbf{W}^1} + S(3) = \frac{u(3)}{(\mathbf{W}^2)^3} + \frac{u(3)}{(\mathbf{W}^2)^2} + \frac{u(2)}{\mathbf{W}^2} + u(1) \tag{B.25}$$

$$\begin{aligned} \left.\frac{\partial F}{\partial \mathbf{W}^1}\right|_4 &= \left.\frac{\partial F}{\partial \mathbf{W}^1}\right|_3 + [U(4) \times S(4)]^T \times \frac{\partial^e F}{\partial y(4)} \\ &= -u(1)(t(1)-y(1)) - (u(2) + \mathbf{W}^2 u(1))(t(2)-y(2)) \\ &\quad -(u(3) + \mathbf{W}^2 u(2) + (\mathbf{W}^2)^2 u(1))(t(3)-y(3)) \\ &\quad -(u(3) + \mathbf{W}^2 u(3) + (\mathbf{W}^2)^2 u(2) + (\mathbf{W}^2)^3 u(1))(t(4)-y(4)) \end{aligned} \tag{B.26}$$

This method implies 17 multiplications/divisions and 10 additions. From this example we concluded that this method is not as efficient as proposed in the literature.

The Fast Forward Propagation Method must be applied recursively by using Eq. (3.67), Eq. (3.68) and Eq. (3.65):

$$b(1) = 0 \tag{B.27}$$

$$A(1) = 1 \tag{B.28}$$

$$\left.\frac{\partial F}{\partial y(1)}\right|_1 = \left.\frac{\partial F}{\partial y(1)}\right|_0 + \frac{\partial^e F}{\partial y(1)} = -(t(1)-y(1)) \qquad \textbf{(B.29)}$$

$$b(2) = \left(\frac{\partial^e y(2)}{\partial y(1)}\right)^{-1} \times \left(b(1) - \frac{\partial^e F}{\partial y(1)}\right) = \frac{(t(1)-y(1))}{\mathbf{W}^2} \qquad \textbf{(B.30)}$$

$$A(2) = \left(\frac{\partial^e y(2)}{\partial y(1)}\right)^{-1} \times A(1) = \frac{1}{\mathbf{W}^2} \qquad \textbf{(B.31)}$$

$$\left.\frac{\partial F}{\partial y(1)}\right|_2 = \left.\frac{\partial F}{\partial y(1)}\right|_1 + \frac{\partial^e y(2)}{\partial y(1)} \times \frac{\partial^e F}{\partial y(2)} = -(t(1)-y(1)) - \mathbf{W}^2(t(2)-y(2)) \qquad \textbf{(B.32)}$$

$$b(3) = \left(\frac{\partial^e y(3)}{\partial y(2)}\right)^{-1} \times \left(b(2) - \frac{\partial^e F}{\partial y(2)}\right) = \frac{1}{\mathbf{W}^2}\left(\frac{(t(1)-y(1))}{\mathbf{W}^2} + (t(2)-y(2))\right) \qquad \textbf{(B.33)}$$

$$A(3) = \left(\frac{\partial^e y(3)}{\partial y(2)}\right)^{-1} \times A(2) = \frac{1}{(\mathbf{W}^2)^2} \qquad \textbf{(B.34)}$$

$$\begin{aligned}\left.\frac{\partial F}{\partial y(1)}\right|_3 &= \left.\frac{\partial F}{\partial y(1)}\right|_2 + \frac{\partial^e y(3)}{\partial y(2)} \times \frac{\partial^e y(2)}{\partial y(1)} \times \frac{\partial^e F}{\partial y(2)} \\ &= -(t(1)-y(1)) - \mathbf{W}^2(t(2)-y(2)) - (\mathbf{W}^2)^2(t(3)-y(3))\end{aligned} \qquad \textbf{(B.35)}$$

$$b(4) = \left(\frac{\partial^e y(4)}{\partial y(3)}\right)^{-1} \times \left(b(3) - \frac{\partial^e F}{\partial y(3)}\right) = \frac{1}{\mathbf{W}^2}\left(\frac{1}{\mathbf{W}^2}\left(\frac{(t(1)-y(1))}{\mathbf{W}^2} + (t(2)-y(2))\right) + (t(3)-y(3))\right) \qquad \textbf{(B.36)}$$

$$A(4) = \left(\frac{\partial^e y(4)}{\partial y(3)}\right)^{-1} \times A(3) = \frac{1}{(\mathbf{W}^2)^3} \qquad \textbf{(B.37)}$$

$$\begin{aligned}\left.\frac{\partial F}{\partial y(1)}\right|_4 &= \left.\frac{\partial F}{\partial y(1)}\right|_3 + \frac{\partial^e y(4)}{\partial y(3)} \times \frac{\partial^e y(3)}{\partial y(2)} \times \frac{\partial^e y(2)}{\partial y(1)} \times \frac{\partial^e F}{\partial y(2)} \\ &= -(t(1)-y(1)) - \mathbf{W}^2(t(2)-y(2)) - (\mathbf{W}^2)^2(t(3)-y(3)) - (\mathbf{W}^2)^3(t(4)-y(4))\end{aligned} \qquad \textbf{(B.38)}$$

We finally can apply Eq. (3.69) to obtain the gradient respect to $\mathbf{W}^1$:

$$
\begin{aligned}
\frac{\partial F}{\partial \mathbf{W}^1} &= \sum_{t=1}^{4}\left[\frac{\partial^e y(t)}{\partial \mathbf{W}^1}\right]^T \times b(t) + \sum_{t=1}^{4}\left[\frac{\partial^e y(t)}{\partial \mathbf{W}^1}\right]^T \times A(t) \times \left.\frac{\partial F}{\partial y(1)}\right|_Q \\
&= \cancel{u(2)\frac{(t(1)-y(1))}{\mathbf{W}^2}} + \cancel{u(3)\frac{(t(1)-y(1))}{(\mathbf{W}^2)^2}} + \cancel{u(3)\frac{(t(2)-y(2))}{\mathbf{W}^2}} + \cancel{u(4)\frac{(t(1)-y(1))}{(\mathbf{W}^2)^3}} \\
&\quad + \cancel{u(4)\frac{(t(2)-y(2))}{(\mathbf{W}^2)^2}} + \cancel{u(4)\frac{(t(3)-y(3))}{\mathbf{W}^2}} - u(1)(t(1)-y(1)) - u(1)\mathbf{W}^2(t(2)-y(2)) \\
&\quad - u(1)(\mathbf{W}^2)^2(t(3)-y(3)) - u(1)(\mathbf{W}^2)^3(t(4)-y(4)) - \cancel{\frac{u(2)(t(1)-y(1))}{\mathbf{W}^2}} \\
&\quad - u(2)(t(2)-y(2)) - u(2)\mathbf{W}^2(t(3)-y(3)) - u(2)(\mathbf{W}^2)^2(t(4)-y(4)) \\
&\quad - \cancel{\frac{u(3)(t(1)-y(1))}{(\mathbf{W}^2)^2}} - \cancel{\frac{u(3)(t(2)-y(2))}{\mathbf{W}^2}} - u(3)(t(3)-y(3)) - u(3)\mathbf{W}^2(t(4)-y(4)) \\
&\quad - \cancel{\frac{u(4)(t(1)-y(1))}{(\mathbf{W}^2)^3}} - \cancel{\frac{u(4)(t(2)-y(2))}{(\mathbf{W}^2)^2}} - \cancel{\frac{u(4)(t(3)-y(3))}{\mathbf{W}^2}} - u(4)(t(4)-y(4)) \\
&= -u(1)(t(1)-y(1)) - u(1)\mathbf{W}^2(t(2)-y(2)) - u(1)(\mathbf{W}^2)^2(t(3)-y(3)) \\
&\quad - u(1)(\mathbf{W}^2)^3(t(4)-y(4)) - u(2)(t(2)-y(2)) - u(2)\mathbf{W}^2(t(3)-y(3)) \\
&\quad - u(2)(\mathbf{W}^2)^2(t(4)-y(4)) - u(3)(t(3)-y(3)) - u(3)\mathbf{W}^2(t(4)-y(4)) - u(4)(t(4)-y(4))
\end{aligned}
\tag{B.39}
$$

This method implies 24 multiplications/divisions and 14 additions.

The Block Update method is intended to obtain the gradient in multiple stages. For the example presented here we will divide the sequence in two segments of two samples each. First we apply the regular BTT algorithm from *t=1* to *t=2*:

$$
\frac{\partial F}{\partial y(2)} = \frac{\partial^e F}{\partial y(2)} + \frac{\partial^e y(3)}{\partial y(2)} \times \frac{\partial F}{\partial y(3)} = -(t(2)-y(2))
\tag{B.40}
$$

$$
\frac{\partial F}{\partial y(1)} = \frac{\partial^e F}{\partial y(1)} + \frac{\partial^e y(2)}{\partial y(1)} \times \frac{\partial F}{\partial y(2)} = -(t(1)-y(1)) - \mathbf{W}^2(t(2)-y(2))
\tag{B.41}
$$

By substitution in Eq. (3.52):

$$
\begin{aligned}
\left.\frac{\partial F}{\partial \mathbf{W}^1}\right|_2 &= \sum_{t=1}^{2}\left[\frac{\partial^e y(t)}{\partial \mathbf{W}^1}\right]^T \times \frac{\partial F}{\partial y(t)} \\
&= -u(1)(t(1)-y(1)) - u(1)\mathbf{W}^2(t(2)-y(2)) - u(2)(t(2)-y(2))
\end{aligned}
\tag{B.42}
$$

We can now apply Eq. (3.83):

$$
G(4) = \sum_{t=3}^{4}\left[\frac{\partial^e y(t)}{\partial \mathbf{W}^1}\right]^T \times \mathbf{\Gamma}(t, 4) + \mathbf{Q}_i(2) \times \mathbf{\Gamma}(3, 4)
\tag{B.43}
$$

where:

$$
\mathbf{\Gamma}(4, 4) = \frac{\partial^e F}{\partial y(4)} + \frac{\partial^e y(5)}{\cancel{\partial y(4)}}\cancel{\times \mathbf{\Gamma}(5, Q)} = -(t(4)-y(4))
\tag{B.44}
$$

$$\mathbf{\Gamma}(3,4) = \frac{\partial^e F}{\partial y(3)} + \frac{\partial^e y(4)}{\partial y(3)} \times \mathbf{\Gamma}(4, Q) = -(t(3)-y(3)) - \mathbf{W}^2(t(3)-y(3)) \qquad \textbf{(B.45)}$$

$$\begin{aligned} \mathbf{Q}_i(2) &= \sum_{t=1}^{Q-N} \left[\frac{\partial^e y(t)}{\partial \mathbf{W}^1}\right]^T \times \frac{\partial^e y(t+1)}{\partial y(t)} \times \frac{\partial^e y(t+2)}{\partial y(t+1)} \times \ldots \times \frac{\partial^e y(Q-N+1)}{\partial y(Q-N)} \\ &= \left[\frac{\partial^e y(1)}{\partial \mathbf{W}^1}\right]^T \times \frac{\partial^e y(2)}{\partial y(1)} \times \frac{\partial^e y(3)}{\partial y(2)} + \left[\frac{\partial^e y(2)}{\partial \mathbf{W}^1}\right]^T \times \frac{\partial^e y(3)}{\partial y(2)} \\ &= u(1)(\mathbf{W}^2)^2 + u(2)\mathbf{W}^2 \end{aligned} \qquad \textbf{(B.46)}$$

by substitution in Eq. (B.43):

$$\begin{aligned} G(4) = &-u(4)(t(4)-y(4)) - u(3)(t(3)-y(3)) - u(3)\mathbf{W}^2(t(4)-y(4)) \\ &-u(1)(\mathbf{W}^2)^2(t(3)-y(3)) - u(1)(\mathbf{W}^2)^3(t(4)-y(4)) \\ &-u(2)\mathbf{W}^2(t(3)-y(3)) - u(2)(\mathbf{W}^2)^2(t(4)-y(4)) \end{aligned} \qquad \textbf{(B.47)}$$

We can obtain the final gradient by using the results of Eq. (B.42) and Eq. (B.47) and solving Eq. (3.70) for $G(4)$:

$$\begin{aligned} \left.\frac{\partial F}{\partial \mathbf{W}^1}\right|_4 &= \left.\frac{\partial F}{\partial \mathbf{W}^1}\right|_2 + G(4) \\ &= -u(1)(t(1)-y(1)) - u(1)\mathbf{W}^2(t(2)-y(2)) - u(2)(t(2)-y(2)) \\ &\quad - u(4)(t(4)-y(4)) - u(3)(t(3)-y(3)) - u(3)\mathbf{W}^2(t(4)-y(4)) \\ &\quad - u(1)(\mathbf{W}^2)^2(t(3)-y(3)) - u(1)(\mathbf{W}^2)^3(t(4)-y(4)) \\ &\quad - u(2)\mathbf{W}^2(t(3)-y(3)) - u(2)(\mathbf{W}^2)^2(t(4)-y(4)) \end{aligned} \qquad \textbf{(B.48)}$$

To obtain the gradient of $\mathbf{W}^1$ using the Block Update method we required 10 multiplications and 10 additions.

The required operations for each method are sumarized on Table 3.1 on page 28.

Appendix C

EXTRA FILTER DESIGN APPLICATION

Let us modify the dynamic neural network shown in section 7.2. on page 78 and use the full cascaded recurrent neural network presented by Li and Haykin [27] as shown in Figure C.1. For this model the intermediate layers (2, 4 and 6) are connected to an additional layer that combines the effect of those layers. Also, the sixth layer is fed back to the first layer.

The Backpropagation order is now 7-6-5-4-3-2-1, so we start in the last layer in the backpropagation order (Layer 7) to get the following equations:

$$\mathbf{S}^{7,7}(t) = \dot{\mathbf{F}}^{7}(\mathbf{n}^{7}(t)) \; ; \; U' = \{7\} \; ; \; E_S(7) = 7$$

$$\mathbf{S}^{7,6}(t) = \mathbf{S}^{7,7}(t)\mathbf{LW}^{7,6}(0)\dot{\mathbf{F}}^{6}(\mathbf{n}^{6}(t))$$

$$\mathbf{S}^{6,6}(t) = \dot{\mathbf{F}}^{6}(\mathbf{n}^{6}(t)) \; ; \; U' = \{7,6\} \; ; \; E_S(6) = 6$$

$$\mathbf{S}^{7,5}(t) = \mathbf{S}^{7,6}(t)\mathbf{LW}^{6,5}(0)\dot{\mathbf{F}}^{5}(\mathbf{n}^{5}(t))$$

$$\mathbf{S}^{6,5}(t) = \mathbf{S}^{6,6}(t)\mathbf{LW}^{6,5}(0)\dot{\mathbf{F}}^{5}(\mathbf{n}^{5}(t))$$

$$\mathbf{S}^{7,4}(t) = \mathbf{S}^{7,7}(t)\mathbf{LW}^{7,4}(0)\dot{\mathbf{F}}^{4}(\mathbf{n}^{4}(t))$$

$$\mathbf{S}^{4,4}(t) = \dot{\mathbf{F}}^{4}(\mathbf{n}^{4}(t)) \; ; \; U' = \{7,6,4\} \; ; \; E_S(4) = 4$$

$$\mathbf{S}^{7,3}(t) = \mathbf{S}^{7,4}(t)\mathbf{LW}^{4,3}(0)\dot{\mathbf{F}}^{3}(\mathbf{n}^{3}(t))$$

$$\mathbf{S}^{4,3}(t) = \mathbf{S}^{4,4}(t)\mathbf{LW}^{4,3}(0)\dot{\mathbf{F}}^{3}(\mathbf{n}^{3}(t))$$

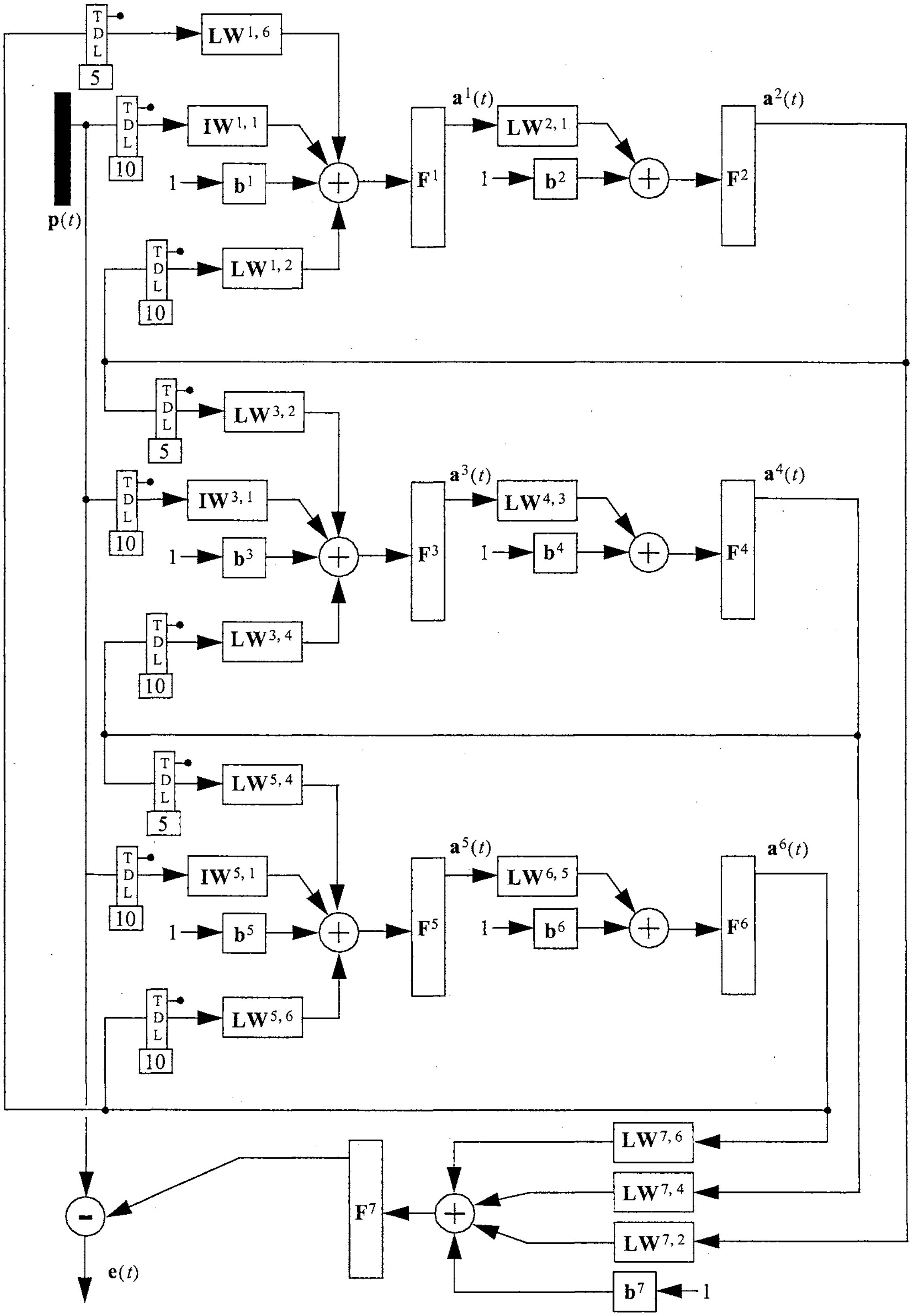

Figure C.1: *Full Cascaded Recurrent Neural Network*

$$\mathbf{S}^{7,2}(t) = \mathbf{S}^{7,7}(t)\mathbf{LW}^{7,2}(0)\dot{\mathbf{F}}^{2}(\mathbf{n}^{2}(t))$$

$$\mathbf{S}^{2,2}(t) = \dot{\mathbf{F}}^{2}(\mathbf{n}^{2}(t))\,;\ U' = \{7,6,4,2\}\ ;\ E_S(2) = 2$$

$$\mathbf{S}^{7,1}(t) = \mathbf{S}^{7,2}(t)\mathbf{LW}^{2,1}(0)\dot{\mathbf{F}}^{1}(\mathbf{n}^{1}(t))$$

$$\mathbf{S}^{2,1}(t) = \mathbf{S}^{2,2}(t)\mathbf{LW}^{2,1}(0)\dot{\mathbf{F}}^{1}(\mathbf{n}^{1}(t))$$

To compute the dynamic derivatives of the performance function respect to the outputs of the system, we fount that the only explicit derivative is respect to the output 7

$$\frac{\partial^e F}{\partial \mathbf{a}^7(t)} = -2(\mathbf{p}(t) - \mathbf{a}^7(t))$$

so the dynamic equations are:

$$\frac{\partial F}{\partial \mathbf{a}^7(t)} = \frac{\partial^e F}{\partial \mathbf{a}^7(t)} + \sum_{d \in DL_{5,6}} \mathbf{LW}^{5,6}(d)^T \mathbf{S}^{6,5}(t+d)^T \frac{\partial F}{\partial \mathbf{a}^6(t+d)}$$

$$\begin{aligned}\frac{\partial F}{\partial \mathbf{a}^6(t)} = {} & \sum_{d \in DL_{1,6}} \mathbf{LW}^{1,6}(d)^T \mathbf{S}^{7,1}(t+d)^T \frac{\partial F}{\partial \mathbf{a}^7(t+d)} + \sum_{d \in DL_{1,6}} \mathbf{LW}^{1,6}(d)^T \mathbf{S}^{2,1}(t+d)^T \frac{\partial F}{\partial \mathbf{a}^2(t+d)} \\ & + \sum_{d \in DL_{5,6}} \mathbf{LW}^{5,6}(d)^T \mathbf{S}^{7,5}(t+d)^T \frac{\partial F}{\partial \mathbf{a}^7(t+d)} + \sum_{d \in DL_{5,6}} \mathbf{LW}^{5,6}(d)^T \mathbf{S}^{6,5}(t+d)^T \frac{\partial F}{\partial \mathbf{a}^6(t+d)}\end{aligned}$$

$$\begin{aligned}\frac{\partial F}{\partial \mathbf{a}^4(t)} = {} & \sum_{d \in DL_{3,4}} \mathbf{LW}^{3,4}(d)^T \mathbf{S}^{7,3}(t+d)^T \frac{\partial F}{\partial \mathbf{a}^7(t+d)} + \sum_{d \in DL_{3,4}} \mathbf{LW}^{3,4}(d)^T \mathbf{S}^{4,3}(t+d)^T \frac{\partial F}{\partial \mathbf{a}^4(t+d)} \\ & + \sum_{d \in DL_{5,4}} \mathbf{LW}^{5,4}(d)^T \mathbf{S}^{7,5}(t+d)^T \frac{\partial F}{\partial \mathbf{a}^7(t+d)} + \sum_{d \in DL_{5,4}} \mathbf{LW}^{5,4}(d)^T \mathbf{S}^{6,5}(t+d)^T \frac{\partial F}{\partial \mathbf{a}^6(t+d)}\end{aligned}$$

$$\begin{aligned}\frac{\partial F}{\partial \mathbf{a}^2(t)} = {} & \sum_{d \in DL_{1,2}} \mathbf{LW}^{1,2}(d)^T \mathbf{S}^{7,1}(t+d)^T \frac{\partial F}{\partial \mathbf{a}^7(t+d)} + \sum_{d \in DL_{1,2}} \mathbf{LW}^{1,2}(d)^T \mathbf{S}^{2,1}(t+d)^T \frac{\partial F}{\partial \mathbf{a}^2(t+d)} \\ & + \sum_{d \in DL_{3,2}} \mathbf{LW}^{3,2}(d)^T \mathbf{S}^{7,3}(t+d)^T \frac{\partial F}{\partial \mathbf{a}^7(t+d)} + \sum_{d \in DL_{3,2}} \mathbf{LW}^{3,2}(d)^T \mathbf{S}^{4,3}(t+d)^T \frac{\partial F}{\partial \mathbf{a}^4(t+d)}\end{aligned}$$

For all layers we have

$$\mathbf{d}^7(t) = [\mathbf{S}^{7,7}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^7(t)}\ ;\ \mathbf{d}^6(t) = [\mathbf{S}^{7,6}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^7(t)} + [\mathbf{S}^{6,6}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^6(t)}$$

$$\mathbf{d}^5(t) = [\mathbf{S}^{7,5}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^7(t)} + [\mathbf{S}^{6,5}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^6(t)}$$

$$\mathbf{d}^4(t) = [\mathbf{S}^{7,4}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^7(t)} + [\mathbf{S}^{4,4}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^4(t)}$$

$$\mathbf{d}^3(t) = [\mathbf{S}^{7,3}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^7(t)} + [\mathbf{S}^{4,3}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^4(t)}$$

$$\mathbf{d}^2(t) = [\mathbf{S}^{7,2}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^7(t)} + [\mathbf{S}^{2,2}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^2(t)}$$

$$\mathbf{d}^1(t) = [\mathbf{S}^{7,1}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^7(t)} + [\mathbf{S}^{2,1}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^2(t)}$$

The previous process is repeated for each sample time in the training set.

Finally we can obtain the gradient for each weight and bias by

$$\frac{\partial F}{\partial \mathbf{b}^7} = \sum_{t=1}^{Q} \mathbf{d}^7(t) ; \frac{\partial F}{\partial \mathbf{LW}^{7,6}(d)} = \sum_{t=1}^{Q} \mathbf{d}^7(t) \times [\mathbf{a}^6(t-d)]^T ; \frac{\partial F}{\partial \mathbf{LW}^{7,4}(d)} = \sum_{t=1}^{Q} \mathbf{d}^7(t) \times [\mathbf{a}^4(t-d)]^T$$

$$\frac{\partial F}{\partial \mathbf{LW}^{7,2}(d)} = \sum_{t=1}^{Q} \mathbf{d}^7(t) \times [\mathbf{a}^2(t-d)]^T ; \frac{\partial F}{\partial \mathbf{b}^6} = \sum_{t=1}^{Q} \mathbf{d}^6(t) ; \frac{\partial F}{\partial \mathbf{LW}^{6,5}(d)} = \sum_{t=1}^{Q} \mathbf{d}^6(t) \times [\mathbf{a}^5(t-d)]^T$$

$$\frac{\partial F}{\partial \mathbf{b}^5} = \sum_{t=1}^{Q} \mathbf{d}^5(t) ; \frac{\partial F}{\partial \mathbf{IW}^{5,1}(d)} = \sum_{t=1}^{Q} \mathbf{d}^5(t) \times [\mathbf{p}^1(t-d)]^T ; \frac{\partial F}{\partial \mathbf{LW}^{5,4}(d)} = \sum_{t=1}^{Q} \mathbf{d}^5(t) \times [\mathbf{a}^4(t-d)]^T$$

$$\frac{\partial F}{\partial \mathbf{LW}^{5,6}(d)} = \sum_{t=1}^{Q} \mathbf{d}^5(t) \times [\mathbf{a}^6(t-d)]^T ; \frac{\partial F}{\partial \mathbf{b}^4} = \sum_{t=1}^{Q} \mathbf{d}^4(t) ; \frac{\partial F}{\partial \mathbf{LW}^{4,3}(d)} = \sum_{t=1}^{Q} \mathbf{d}^4(t) \times [\mathbf{a}^3(t-d)]^T$$

$$\frac{\partial F}{\partial \mathbf{b}^3} = \sum_{t=1}^{Q} \mathbf{d}^3(t) ; \frac{\partial F}{\partial \mathbf{IW}^{3,1}(d)} = \sum_{t=1}^{Q} \mathbf{d}^3(t) \times [\mathbf{p}^1(t-d)]^T ; \frac{\partial F}{\partial \mathbf{LW}^{3,2}(d)} = \sum_{t=1}^{Q} \mathbf{d}^3(t) \times [\mathbf{a}^2(t-d)]^T$$

$$\frac{\partial F}{\partial \mathbf{LW}^{3,4}(d)} = \sum_{t=1}^{Q} \mathbf{d}^3(t) \times [\mathbf{a}^4(t-d)]^T ; \frac{\partial F}{\partial \mathbf{b}^2} = \sum_{t=1}^{Q} \mathbf{d}^2(t) ; \frac{\partial F}{\partial \mathbf{LW}^{2,1}(d)} = \sum_{t=1}^{Q} \mathbf{d}^2(t) \times [\mathbf{a}^1(t-d)]^T$$

$$\frac{\partial F}{\partial \mathbf{b}^1} = \sum_{t=1}^{Q} \mathbf{d}^1(t) ; \frac{\partial F}{\partial \mathbf{IW}^{1,1}(d)} = \sum_{t=1}^{Q} \mathbf{d}^1(t) \times [\mathbf{p}^1(t-d)]^T ; \frac{\partial F}{\partial \mathbf{LW}^{1,2}(d)} = \sum_{t=1}^{Q} \mathbf{d}^1(t) \times [\mathbf{a}^2(t-d)]^T$$

Again, after the network was trained, it was used to predict blood concentration. Figure C.2 is a plot of the errors between actual and predicted signals. Using the last network we were able to reduce the prediction error by half.

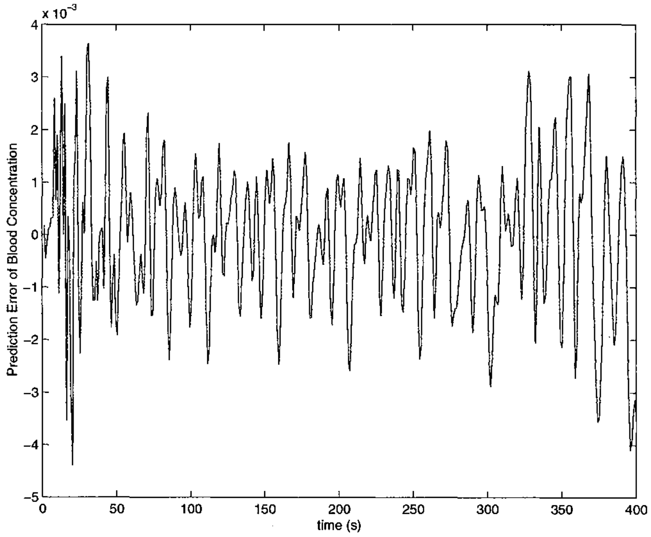

Figure C.2: *Errors for LDRN with Full Dynamic Training*

Appendix D

NEURAL NETWORK EXAMPLES

This chapter contains the neural network examples used to test the dynamic training algorithms. Each circle represents a layer as described in section 2.3 on page 8 and section 2.4 on page 9. The tapped delay line (see Figure 2.4 on page 10) represents a default value that may change for different experiments.

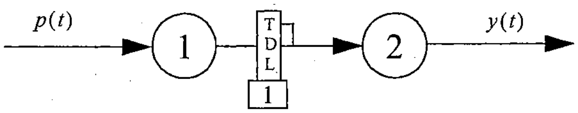

Figure D.1: *Network 1: 2-layer LDDN with* [0 1] *delay between layers 1 and 2.*

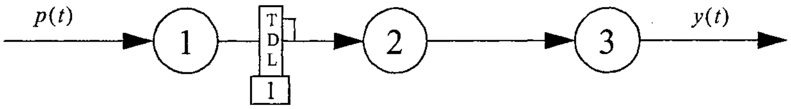

Figure D.2: *Network 2: 3-layer LDDN with* [0 1] *delay between layers 1 and 2.*

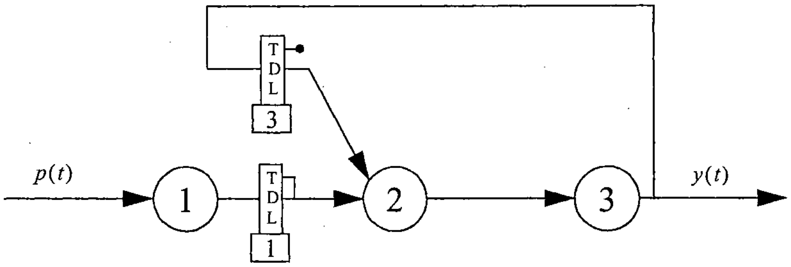

Figure D.3: *Network 3: 3-layer LDDN with* [0 1] *TDL from layer 1 to 2 and* [1 2 3] *TDL from layer 3 to 2.*

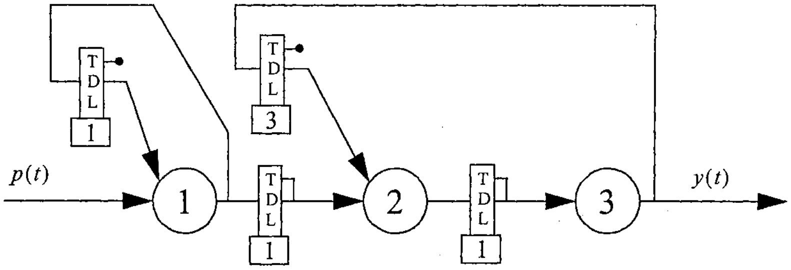

Figure D.4: *Network 4: 3-layer LDDN example.*

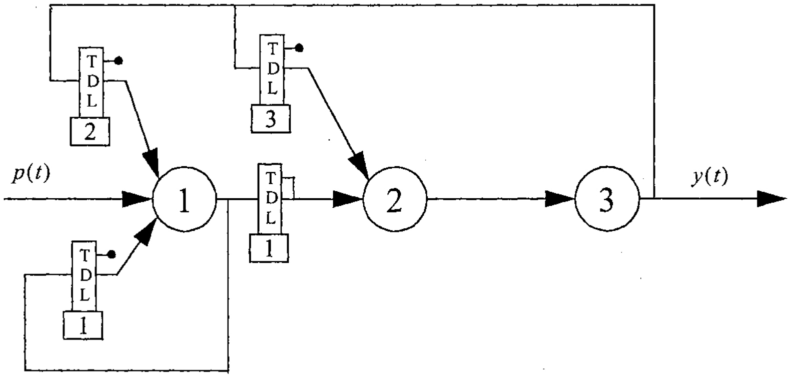

Figure D.5: *Network 5: 3-layer LDDN.*

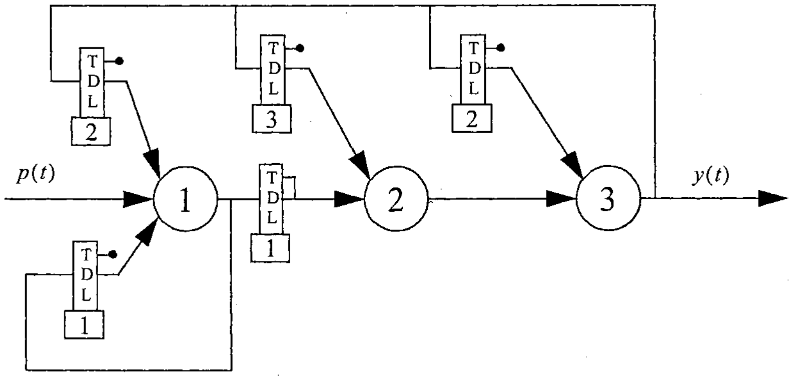

Figure D.6: *Network 6: 3-layer LDDN.*

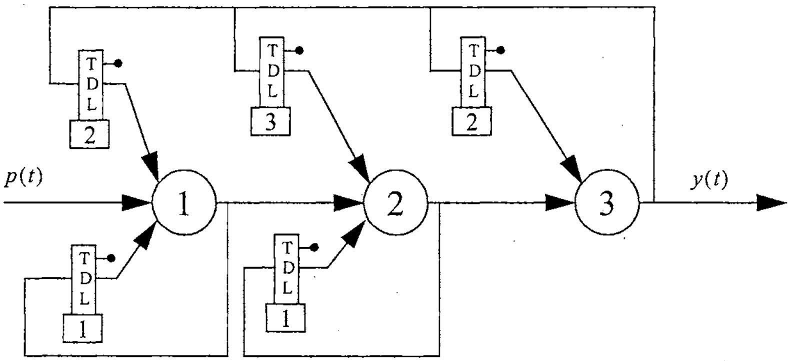

Figure D.7: *Network 7: 3-layer LDDN.*

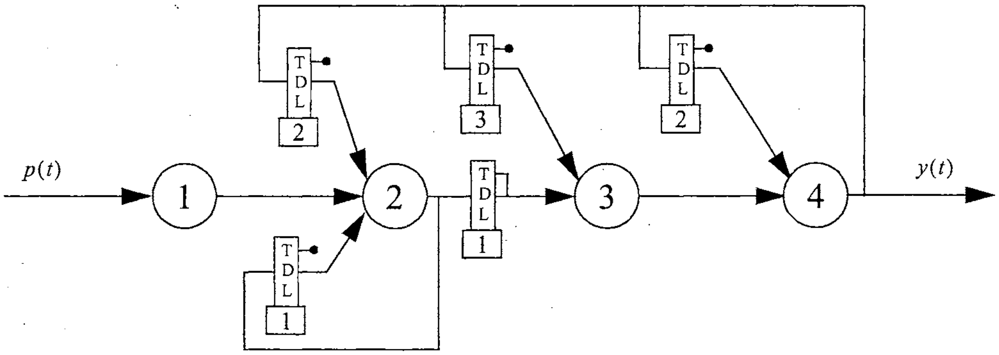

Figure D.8: *Network 8: 4-layer LDDN.*

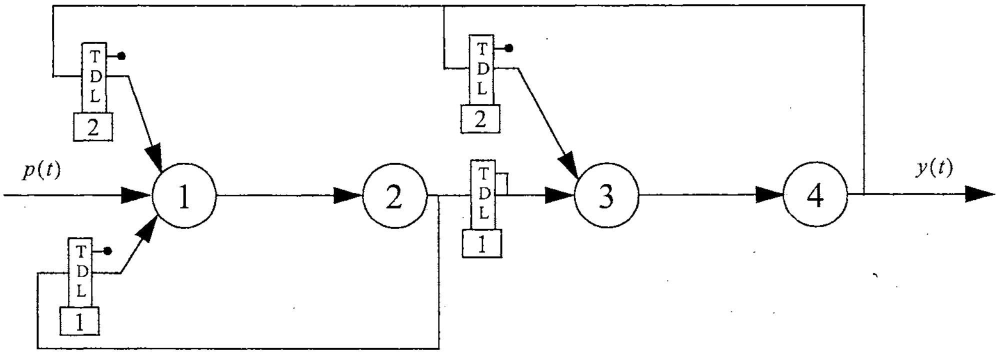

Figure D.9: *Network 9: 4-layer LDDN. This model resembles the Model Reference Controller.*

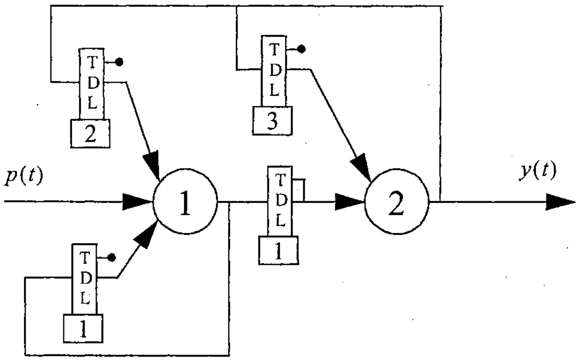

Figure D.10: *Network 10: 2-layer LDDN.*

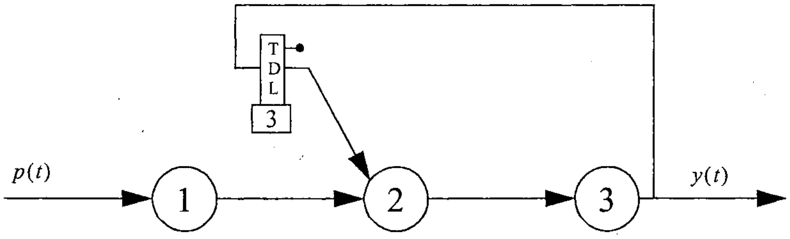

Figure D.11: *Network 11: 3-layer LDDN with* [1 2 3] *feedback from layer 3 to 2.*

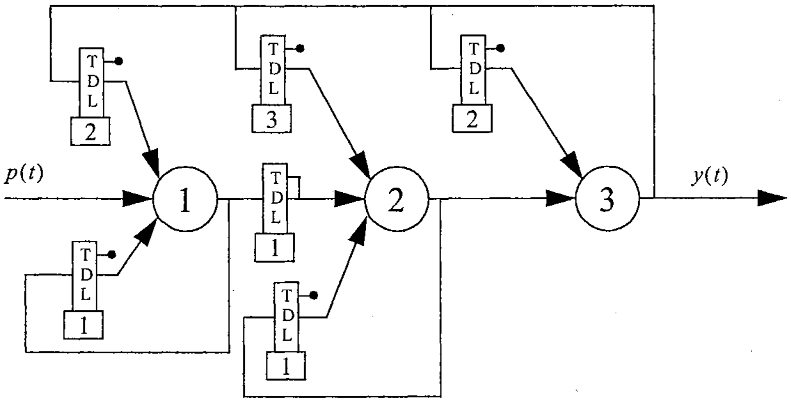

Figure D.12: *Network 12: 3-layer LDDN.*

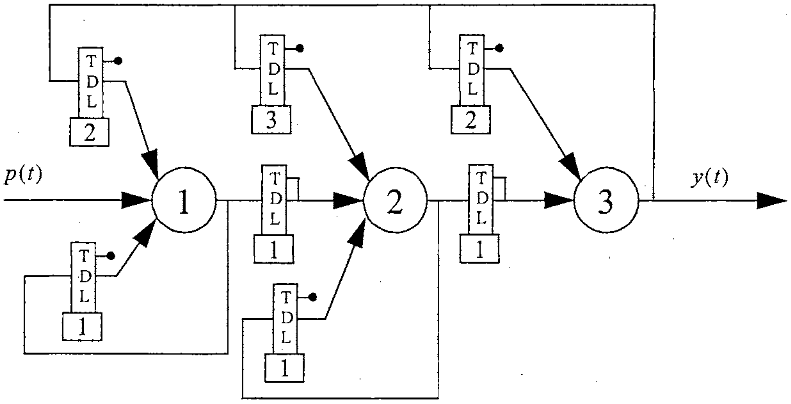

Figure D.13: *Network 13: 3-layer LDDN.*

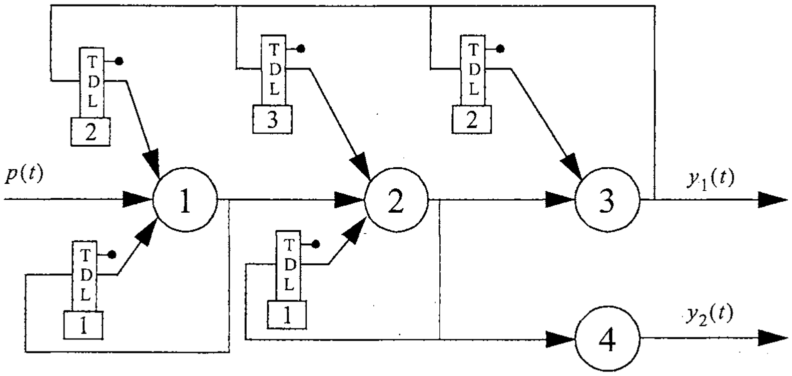

Figure D.14: *Network 14: 4-layer LDDN.*

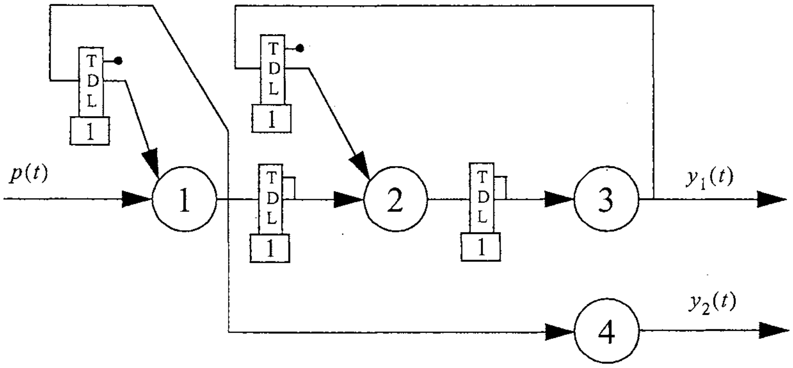

Figure D.15: *Network 15: 4-layer LDDN.*

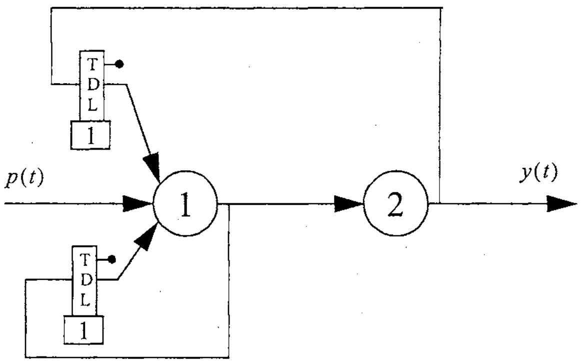

Figure D.16: *Network 16: 2-layer LDDN.*

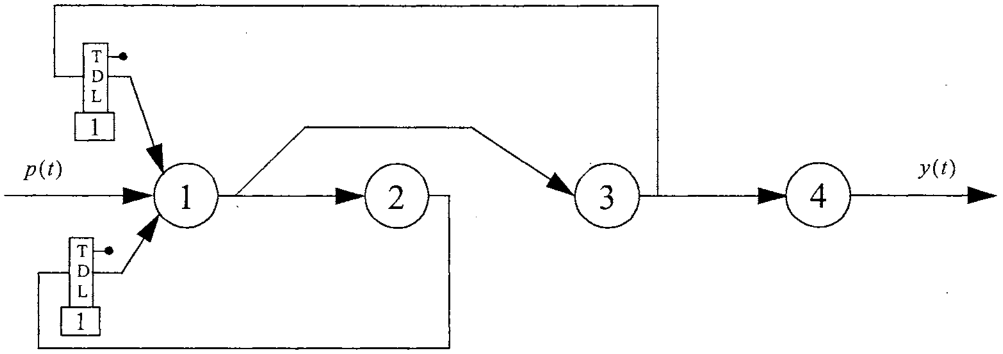

Figure D.17: *Network 17: 4-layer LDDN.*

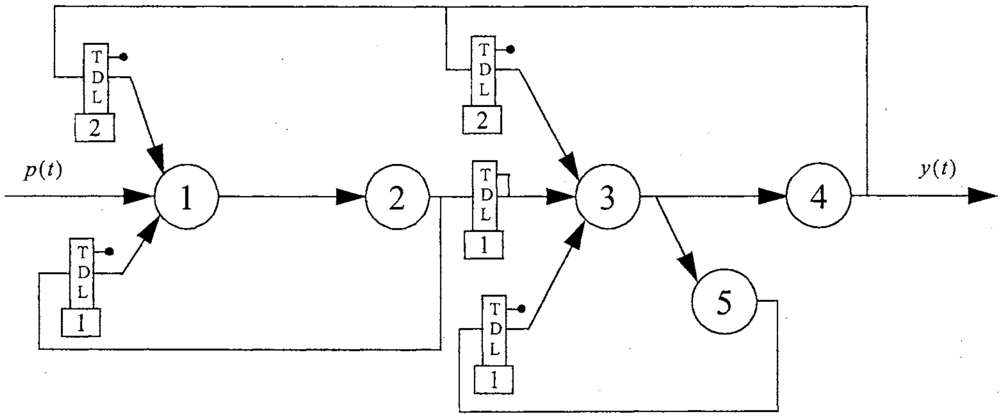

Figure D.18: *Network 18: 5-layer LDDN.*

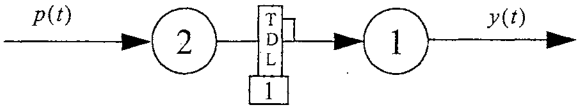

Figure D.19: *Network 19: 2-layer LDDN similar to Network 1 (Figure B.1) with layer numbers in reverse order.*

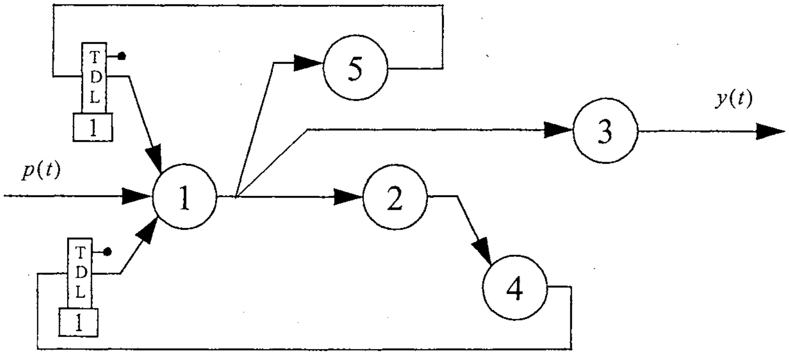

Figure D.20: *Network 20: 5-layer LDDN.*

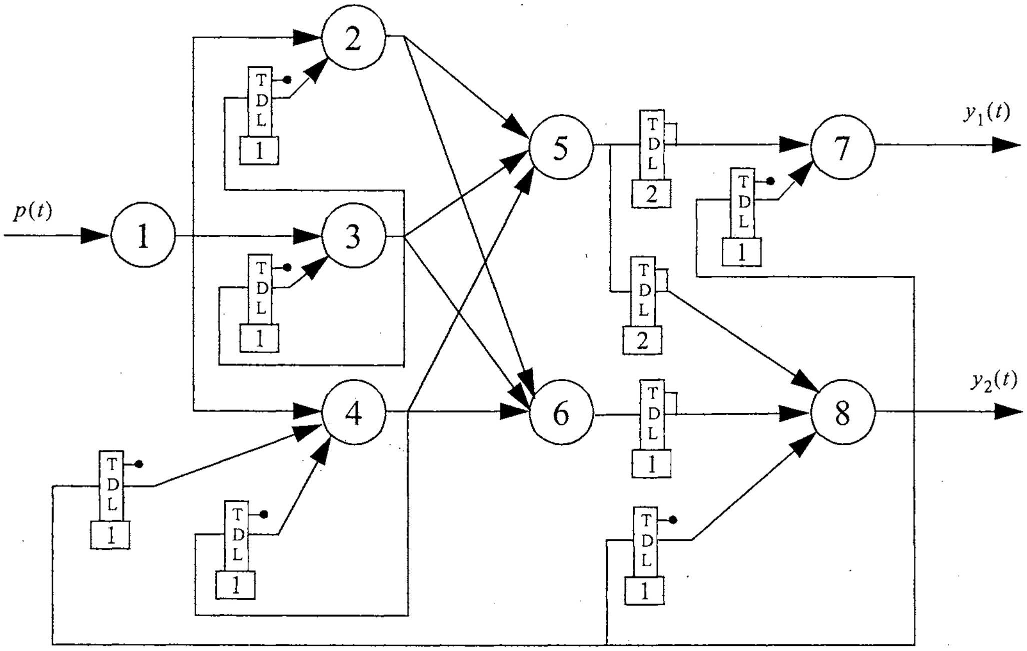

Figure D.21: *Network 21: 8-layer LDDN with two outputs connected to layers 7 and 8.*

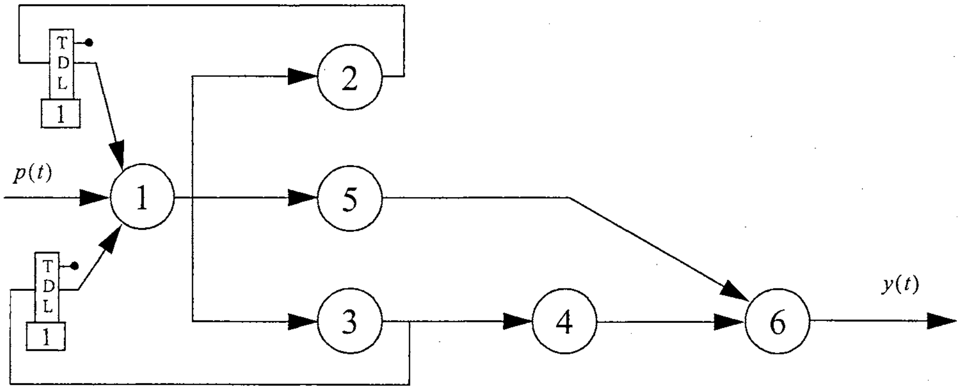

Figure D.22: *Network 22: 6-layer LDDN with unitary feedback from layers 2 and 3 to layer 1.*

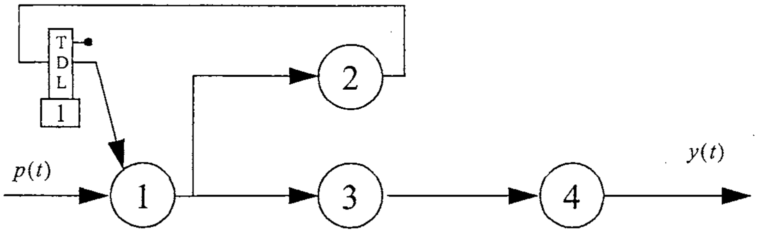

Figure D.23: *Network 23: 4-layer LDDN with unitary feedback from layer 2 to layer 1.*

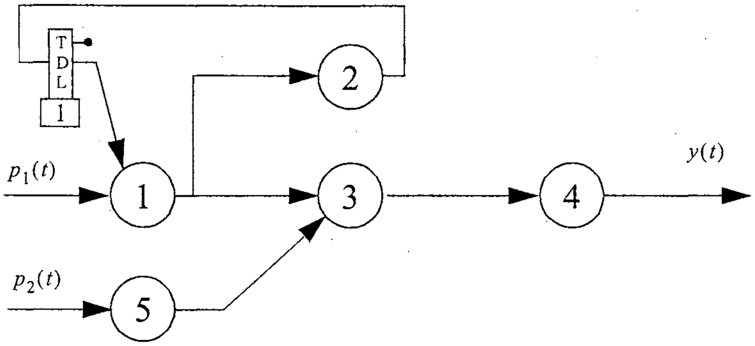

Figure D.24: *Network 24: 5-layer LDDN with two inputs connected to layers 1 and 5.*

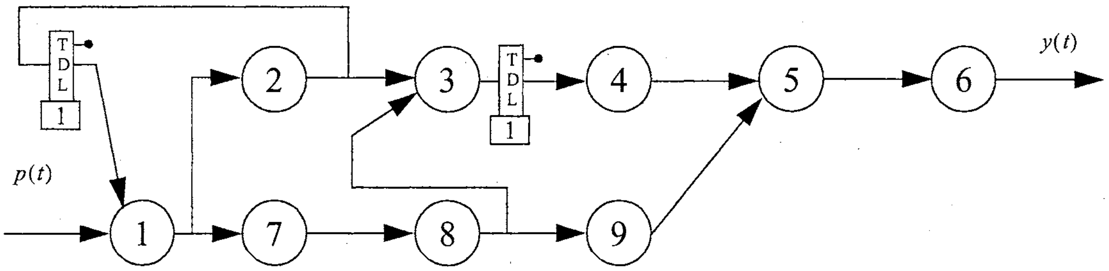

Figure D.25: *Network 25: 9-layer LDDN with output connected to layer 6.*

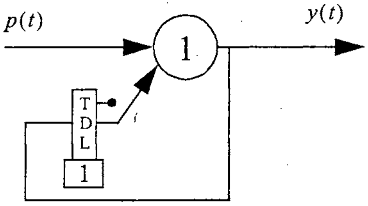

Figure D.26: *Network 26: 1-layer LDDN with unitary feedback to itself.*

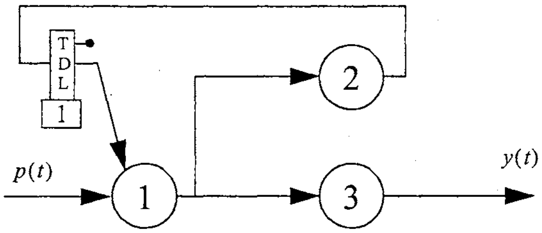

Figure D.27: *Network 27: 3-layer LDDN with unitary feedback from layer 2 to layer 1.*

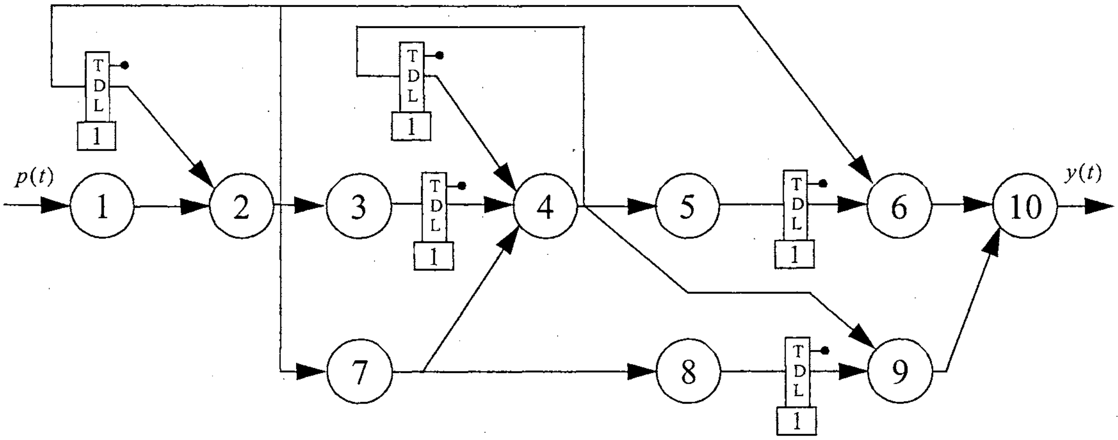

Figure D.28: *Network 28: 10-layer LDDN with multiple delays and feedback.*

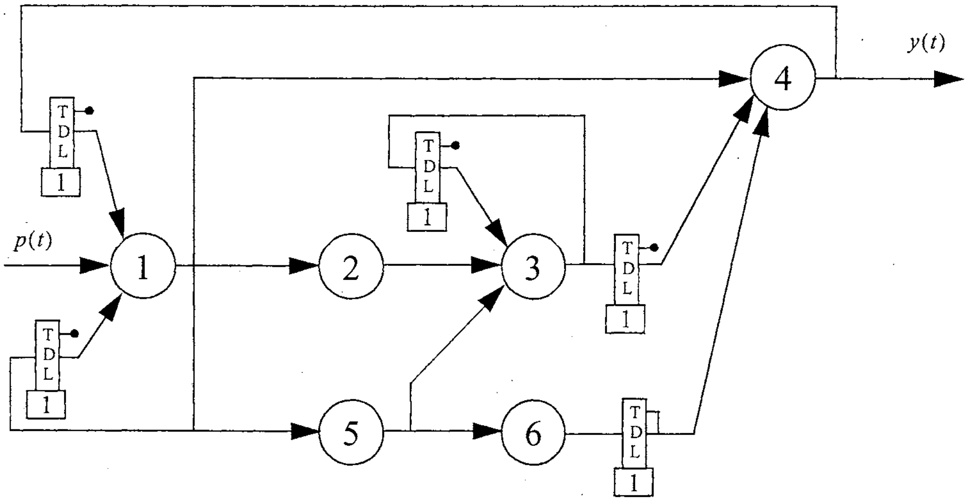

Figure D.29: *Network 29: 6-layer LDDN with multiple TDLs and output connected to layer 4.*

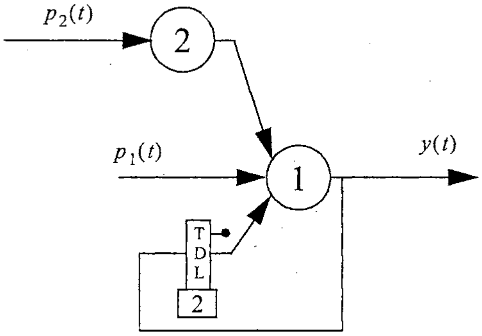

Figure D.30: *Network 30: 2-layer LDDN with two inputs connected to both layers.*

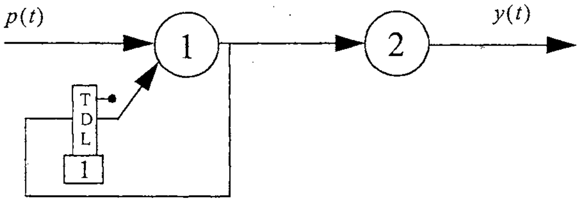

Figure D.31: *Network 31: 2-layer LDDN with unitary feedback from layer 1 to itself.*

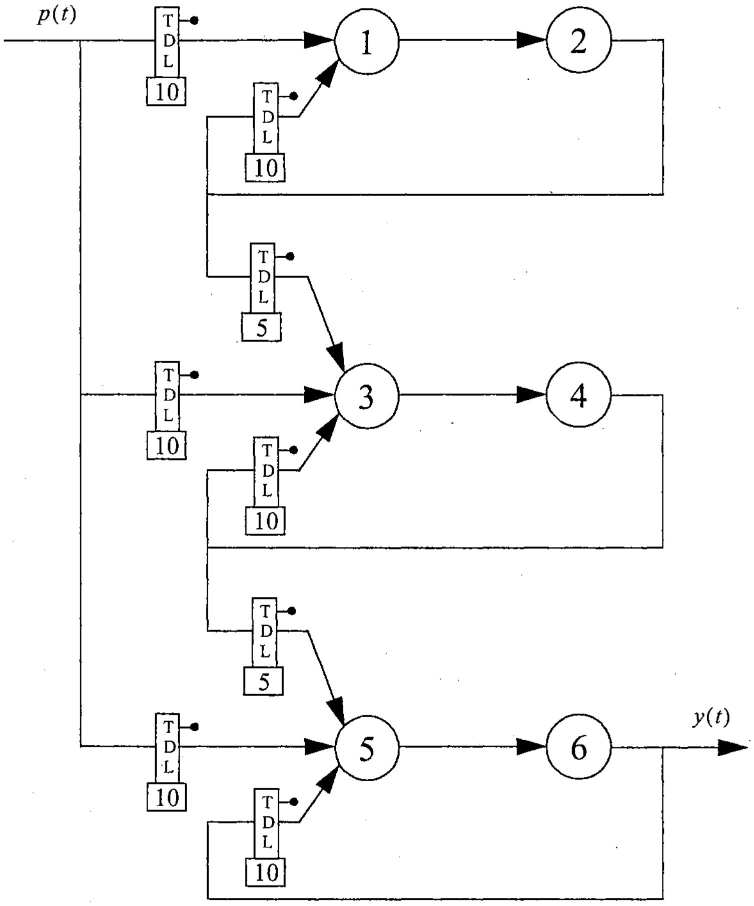

Figure D.32: *Network 32: 6-layer LDDN in modified cascaded recurrent neural network configuration.*

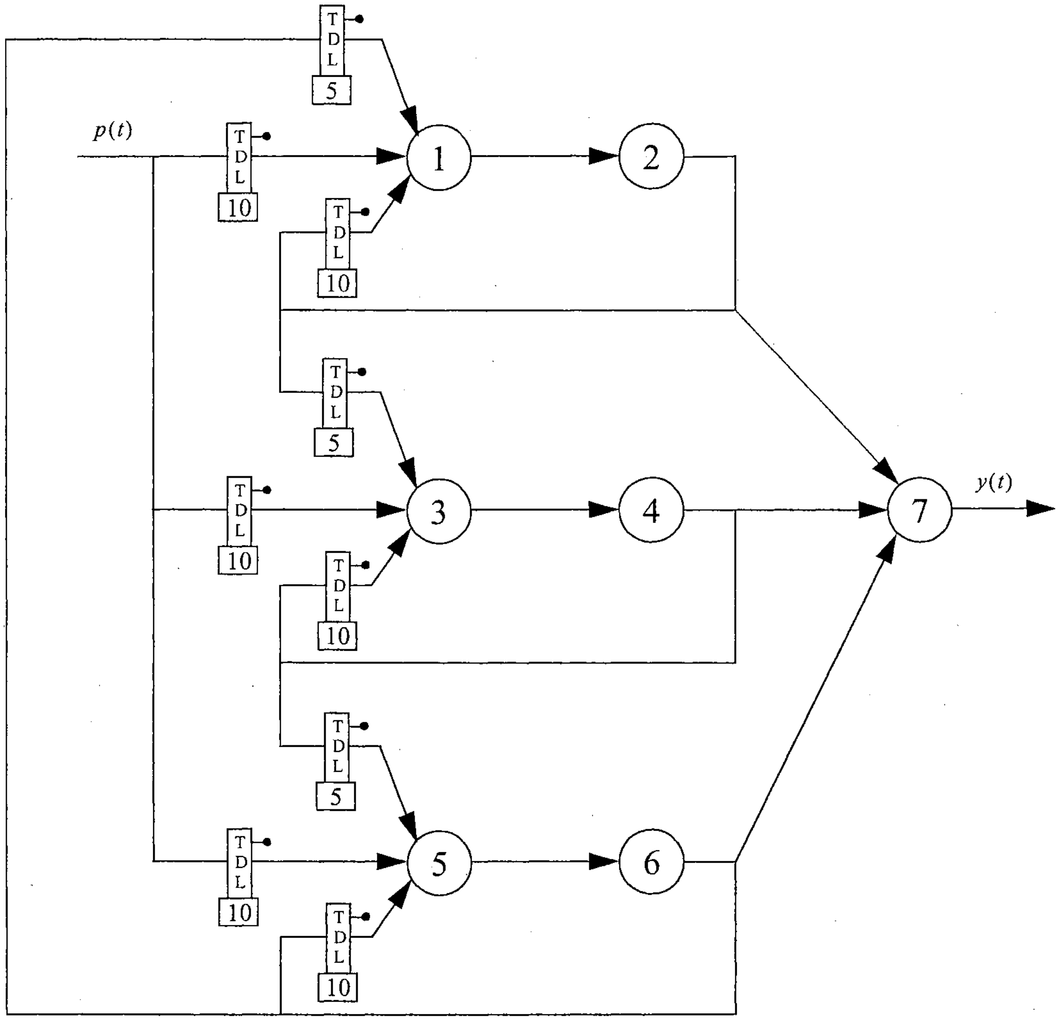

Figure D.33: *Network 33: 7-layer LDDN in full cascaded recurrent neural network configuration.*

VITA 2

Orlando De Jesús

Candidate for the Degree of

Doctor of Philosophy

Thesis: TRAINING GENERAL DYNAMIC NEURAL NETWORKS

Major Field: Electrical Engineering

Biographical:

Personal Data: Born in Caracas, Venezuela, on October 3, 1963, the son of Manuel De Jesús and Lucía N. De Abreu.

Education: Graduated from Creación Guarenas High School, Guarenas, Edo. Miranda, Venezuela, in July 1980; received degrees of Engineer in Electronics (Cum Laude) and Project Management Specialist from Universidad Simón Bolívar, Caracas, Venezuela, in July 1985 and July 1992, respectively, and Master of Science in Electrical and Computer Engineering from Oklahoma State University in December 1998. Completed the requirements for the Doctor of Philosophy degree in Electrical and Computer Engineering at Oklahoma State University in August 2002.

Experience: Employed by AETI C.A., Caracas, Venezuela, as a Research Engineer, R&D Manager and Operations Manager from 1985 to 1996; employed by Oklahoma State University as a Research Assistant from 1997 to present.

Professional Memberships: Institute of Electrical and Electronic Engineers (IEEE), The Instrumentation, Systems and Automation Society (ISA).