NEUROCONTROL USING DYNAMIC LEARNING

BY

WEI-CHUNG YANG

Undergraduate Degree
National Taipei Institute of Technology
Taipei, Taiwan
1975

Master of Engineering
The University of Tulsa
Tulsa, Oklahoma
1982

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
May, 1994

NEUROCONTROL USING DYNAMIC LEARNING

Thesis Approved:

Martin T Hagan
Thesis Adviser

Thomas C. Collins
Dean of the Graduate College

ACKNOWLEDGMENTS

I could not reached at this point of my education without the guidance and help of my adviser and friend Dr. Marty Hagan. I am deeply indebted to him. Thank you Dr. Hagan, my forever teacher. Thank you Marty, my forever friend. My appreciation is also extended to Dr. R. Rhoten, Dr. C. Latino and Dr. G. Young for serving on my committee.

My deepest love and appreciation goes with no doubt to my family, especially my mother who, like every mother in the world, loves her son without conditions. I am also grateful to my wife I-Mei for caring our children, Lon-Lon and Phon-Phon, during my study years.

I dedicate this dissertation to all the people who love, support and help me in the course of my life.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

II. NEURAL NETWORK AND FUNCTION APPROXIMATION 5

Biological Neuron Description 5
Artificial Neuron 7
Multilayer Neural Networks 9
Function Approximation 12
Network Performance and Network Architecture . 13
Test 1 : Function Approximation for the Single Hidden Layer Network 16
Test 2: The Oversized Networks 17
Test 3 : Function Approximation for the Two Hidden Layer Network 18
Test 4 : Comparison between Single and Multilayer Networks 19
Summary 21

III. BACKPROPAGATION AND OPTIMIZATION 22

The Supervised Learning Process 22
Backpropagation 24
Optimization Techniques 27
Steepest Descent Method 28
Marquardt Method 31
Conjugate Directions Method 34
Implementation 36
Forward Computation 36
Backpropagation 37
Optimization 37
Summary 38

Chapter Page

IV. SYSTEM IDENTIFICATION AND CONTROL USING STATIC LEARNING 40

System Representations 41
Nonlinear MA Model 41
Nonlinear AR Model 43
Nonlinear ARMA Model 44
System Identification 44
The Swinging Pendulum System 46
Computer Simulations 47
Feedback Linearization 51
Feedback Linearization with Neural Networks 53
Computer Simulations for Nonlinearity Neural Network 54
Computer Simulations for Feedback Linearization Neural Network 57
Direct Inverse Control 59
AMA Method.......................... 61
Computer Simulations 62
Summary 65

V. FORWARD PERTURBATION 66

Recurrent Networks 66
Dynamic Learning 69
Calculations of True Derivative 71
Forward Perturbation 72
Previous Work on FP 73
Recurrent and Non-recurrent Variables ... 73
Basic Dynamic Learning Models and True Derivative Calculation 75
An Illustrative Example 77
Complex Examples in Dynamic Learning ... 81
Marquardt Optimization Using FP 83
Summary 85

VI. BACKPROPAGATION-THROUGH-TIME.................. 86

Derivative Calculation 86
The BTT Derivative Calculation Equations 87
Calculation of STT 88
An Illustrative Example 89

Chapter Page

Complex Examples in Dynamic Learning 93
Marquardt Optimization 96
Using BTT in the Marquardt Method....... 96
Converting STT to PTT 97
A Comparison of FP and BTT 99

VII. NEUROCONTROL USING DYNAMIC LEARNING............ 103

MRAC in Linear Dynamic Systems 104
Indirect MRAC Using Neural Networks 107
Indirect MRAC Using FP 108
Computer Simulations 110
Simulation 1 : First Order Linear Plant with Linear Controller 112
Simulation 2 : First Order Linear Plant with Nonlinear Controller 115
Simulation 3 : Second Order Linear Plant with Linear Controller 118
Simulation 4 : Second Order Linear Plant with Nonlinear Controller 121
Simulation 5 : Second Order Nonlinear Plant with Nonlinear Controller 124
Summary 130

VIII REAL-TIME EXPERIMENTAL RESULTS 132

The Real Physical System Description 132
Real-Time Control using Static Learning 135
Feedback Linearization 135
Procedure for Training and Real-Time Control 136
The Training and Real-Time Control Results 137
Real Physical System Identification 142
Forward Modeling and Evaluation Methods 142
Procedure for Training and Real-Time Evaluation........................... 145
The Training and Real-Time Evaluation Results 145
Real-Time Control using Dynamic Learning....... 152
MRAC and Plant Model 152
Procedure for Training and Real-Time

Chapter Page
Control 154
The Training and Real-Time Control
Results 155

IX. SECONDARY TRAINING METHODS 160

The ATS Method.............................. 161
The ATS Method vs the General Approach.. 161
Computer Simulation I................... 162
The ATS Method vs Single Set Training 170
Computer Simulation II 170
Comments 174
The TCT/TLI Method......................... 176
Computer Simulation III................. 177
Comments 186

X. SUMMARY AND CONCLUSIONS 187

REFERENCES 191

LIST OF TABLES

Table Page

2.1. Results of Test 1 16

2.2. Results of Test 2 17

2.3. Results of Test 3 19

2.4. Results of Test 4 20

6.1. Comparisons of Computation Time (in CPU Time) between the FP and BTT Algorithms 100

6.2. Comparisons of Computation Time (in Flops) between the FP and BTT Algorithms 102

7.1. The Weights Converging Process for Simulation 1 115

7.2. The Weights Converging Process for Simulation 3 121

LIST OF FIGURES

Figure Page

2.1. A Simplified Biological Neuron 7

2.2. Artificial Neuron Model and Its Symbolic Representation 8

2.3. Typical Activation Functions 9

2.4. A MIMO Two Hidden Layer Network 10

2.5. Recurrent Multilayer Network 11

2.6. The Sample Functions 14

3.1. A MIMO Two Hidden Layer Network 24

3.2. An Illustration of Fixed Step Size Learning 29

3.3. An Illustration of the Adaptive Learning Rate Algorithm 30

4.1. The Tapped Delay Line 42

4.2. A SISO Nonlinear MA Model 42

4.3. A SISO Nonlinear AR Model 43

4.4. A SISO Nonlinear ARMA Model 44

4.5. A Nonlinear SISO System Identification Using a Neural Network 45

4.6. A Swinging Pendulum System 46

4.7. The Learning Curve of the Network NN_p 49

4.8. The Series-Parallel Test Method....................... 50

Figure Page

4.9. The Evaluation of the Network NN_f 51

4.10. The learning curve of the trained Network NN_f 55

4.11. The Evaluation of the Network NN_f 56

4.12. Feedback Linearization Vs Linear Controller (#1) 58

4.13. Feedback Linearization Vs Linear Controller (#2) 59

4.14. The Inverse Modeling 60

4.15. Direct Inverse Control 60

4.16. Modeling Inverse Plant with a Neural Network 62

4.17. The Learning Curve of the Network NN_p^{-1} 63

4.18. The Evaluation Tests of the Network NN_p^{-1} 64

5.1. A MIMO Two Hidden Layer Network 67

5.2. A Recurrent Connection 68

5.3. Recurrent Multilayer Network 69

5.4. Example of a Recurrent Network Application............ 70

5.5. Recurrent and Non-recurrent Variables 74

5.6. Dynamic Learning Model I 75

5.7. Dynamic Learning Model II 77

5.8. A Simple Recurrent Network 77

5.9. First Complex Example for Dynamic Learning 82

5.10. Second Complex Example for Dynamic Learning 83

6.1. Dynamic Learning Model I 88

6.2. Dynamic Learning Model II 89

6.3. A Simple Recurrent Network 90

6.4. First Complex Example for Dynamic Learning 94

6.5. Second Complex Example for Dynamic Learning 95

7.1. Direct Adaptive Control 105

Figure Page

7.2. Indirect Adaptive Control 105

7.3. The Illustration of Adaptive Control 106

7.4. Indirect MRAC Using Neural Networks 108

7.5. The Learning Curve of Case 1 114

7.6. The Learning Curve of Case 2 116

7.7. The Evaluation Tests of Case 2 (#1) 117

7.8. The Evaluation Tests of Case 2 (#2) 118

7.9. The Learning Curve of Case 3 120

7.10. The Learning Curve of Case 4 122

7.11. The Evaluation Tests of Case 4 (#1) 123

7.12. The Evaluation Tests of Case 4 (#2) 124

7.13. The Comparison of Non-training and Training responses 126

7.14. The Evaluation Test after Two Epochs Training 127

7.15. The Learning Curve of Case 5 128

7.16. The Last Epoch Training Result and the Evaluation 129

7.17. Four More Evaluation Tests 130

8.1. The Function Diagram of the Pendulum System 134

8.2. The Network Configuration to Train the NN_f and Control Input Constant C 137

8.3. The Learning Curve of Network NN_f 138

8.4. The tests of the Nonlinearity and Friction of the Physical Pendulum System 139

8.5. The Tests of Moving Pendulum Up in Real-Time 140

8.6. The Tests of Moving Pendulum Down in Real-Time 141

8.7. System Identification of a Real Physical Plant 143

Figure Page

8.8. The Series-Parallel Test Method........................ 143

8.9. The Parallel Test Method 144

8.10. The Learning Curve of the Network NN_p 146

8.11. The Network Identifier Moving-Up Series-Parallel Tests 148

8.12. The Network Identifier Moving-Down Series-Parallel Tests 149

8.13. The Network Identifier Moving-Up Parallel Tests 150

8.14. The Network Identifier Moving-Down Parallel Tests 151

8.15. Indirect MRAC using Neural Network.................. 153

8.16. Modified Indirect MRAC using Neural Network.......... 154

8.17. The Moving-Down Test I for MRAC Controller........... 156

8.18. The Moving-Down Test II for MRAC Controller........... 156

8.19. The Moving-Down Test III for MRAC Controller...........157

8.20. The Moving-Up Test I for MRAC Controller 158

8.21. The Moving-Up Test II for MRAC Controller 158

8.22. The Moving-Up Test III for MRAC Controller 159

9.1. The Learning Curves for the General Approach vs. the ATS Method for Case 1 163

9.2. The Learning Curves for the General Approach vs. the ATS Method for Case 2 164

9.3. The Learning Curves for the General Approach vs. the ATS Method for Case 3 165

9.4. Results of the Parallel Tests for Both Approaches For Case 1 167

9.5. Results of the Parallel Tests for Both Approaches

Figure Page

For Case 2 168

9.6. Results of the Parallel Tests for Both Approaches For Case 3 169

9.7. The Learning Curves for the Single set Training Method vs. the ATS Method for Case 1 171

9.8. The Learning Curves for the Single set Training Method vs. the ATS Method for Case 2 172

9.9. The Learning Curves for the Single set Training Method vs. the ATS Method for Case 3 173

9.10. General Results of the Parallel Tests for Both Approaches for All Three Cases 175

9.11. Learning Curves of One Trajectory Training vs the TCT Method for Case 1 179

9.12. Enlargement of the Learning Curves for Epochs 1901 to 2000 from Figure 9.11.......................... 180

9.13. The Parallel Test Results for the Trained Controllers for Case 1 182

9.14. Learning Curves of One Trajectory Training vs the TCT Method for Case 2 183

9.15. Enlargement of the Learning Curves for Epochs 1901 to 2000 from Figure 9.14.......................... 184

9.16. The Parallel Test Results for the Trained Controllers for Case 2 185

CHAPTER I

INTRODUCTION

This research studies the promise of neural networks in the realm of system identification and control of nonlinear dynamic systems. Two types of supervised learning algorithms for neural networks are described. The first is the static learning algorithm, which can be used for system identification, and also for the control of dynamic systems as well. This research focuses on the second supervised learning algorithm -- dynamic learning. The dynamic learning algorithm can be executed on-line in the training of nonlinear dynamic neurocontrollers. Three control schemes are involved in this research -- feedback linearization, direct inverse control and model reference adaptive control. Feedback linearization and direct inverse control are implemented using static learning. The on-line adaptation, which is the common characteristic shared by both the dynamic learning method and the adaptive control method, makes it natural to combine both the methods together in real-time adaptive neurocontrol applications.

This document contains ten chapters. Starting from the neural network building block, an artificial neuron model, the feedforward multilayer network and the recurrent multilayer network are derived and described in the first part of Chapter 2. The second part of Chapter 2 describes the performance of the neural network as a function approximator. Simulations have been executed to investigate the effects of varying the number of neurons and the number of layers in the network on the performance of the function approximation task.

Chapter 3 describes the learning process that is used for training neural networks to perform function approximation. The basic backpropagation method is discussed first because it is an essential tool in gradient optimization. Then, three optimization techniques -- steepest descent, the Marquardt method and the conjugate direction method -- are presented and discussed. The conclusion of Chapter 3 focuses on implementing the neural network learning process from a programming point of view.

The static learning algorithm is described in the beginning of Chapter 4. Then, it is demonstrated in the system identification of a second order nonlinear dynamic pendulum system. To illustrate that static learning can also be used to train dynamic system controllers, two control schemes -- feedback linearization and direct inverse control -- are used with the static learning algorithm to train the neurocontroller off-line.

For on-line adaptive identification and control applications using neural networks, dynamic learning is described in Chapter 5. For implementing the dynamic learning algorithm, the recurrent multilayer networks are presented first. Then, two known, but not well reported, dynamic learning algorithms -- forward perturbation and backpropagation-through-time -- are introduced. We will describe the backpropagation-through-time algorithm in the next chapter. The forward perturbation algorithm is studied in the second half of Chapter 5. First, the derivative calculation equation for the forward perturbation algorithm is derived. Then a simple illustrative example of the forward perturbation algorithm is given before we implement the algorithm on more complex examples.

As in our discussion of the forward perturbation algorithm in Chapter 5, the backpropagation-through-time discussion will begin with the derivation of the backpropagation-through-time derivative calculation equation. This will be followed by implementation examples and a description of the Marquardt optimization method using backpropagation-through-time. Based on computer simulations, the comparison between forward perturbation and backpropagation-through-time will be given.

In the beginning of Chapter 7 the model reference adaptive control method is presented. Then, it is reconfigured with the neural network models of plant and controller. The neural network controller is trained in a real time adaptive fashion. This is all described as the main subject of the first part of Chapter 6. In the second part of Chapter 7, five computer simulations of model reference adaptive neurocontrol are performed using forward perturbation. All the simulation results were successful, which provides us with a promising tool for dealing with more complex real time nonlinear dynamic systems.

In Chapter 8, after the successes in the computer simulations in Chapter 7, we will attempt to implement the trained neurocontrollers from both static learning and dynamic learning to control a real physical pendulum system in real-time. First, a real-time feedback linearization controller is trained off-line using static learning and training data is collected from the physical system in real-time. The same training data is also used to model the physical pendulum system with a neural network. Then a real-time model reference adaptive controller is trained off-line using dynamic learning and the neural network plant model. The real-time experimental results are given for the controllers and the system model.

Our proposed secondary training methods are described in Chapter 9. In static learning, we will compare a secondary training method to a standard approach. For dynamic learning, since there is no standard approach, we will discuss how the proposed secondary training method works and its sensitivity to parameter variations.

Finally, Chapter 10 summarizes the original contributions and the important results presented and described in this document as the conclusions of this research.

CHAPTER II

NEURAL NETWORK AND FUNCTION APPROXIMATION

The first part of this chapter will introduce neural network architecture, starting with the basic building block -- the neuron. A simplified biological neuron is then presented to lay the biological background for the following description and structure of an artificial neuron model. With the artificial neuron defined, a basic multilayer neural network, called the feedforward network is built up. A variation of the basic multilayer network, called the recurrent network, is introduced but will be discussed in Chapter 5. The subject of the second part of this chapter is the performance of the neural network as a function approximator. An effort has been made to investigate the effects of the number of neurons and the number of layers in the network on performing the function approximation task.

Biological Neuron Description

The human brain's mechanism of information processing and decision making has been analyzed not only from the point of view of biology but also of mathematics and engineering. It is recognized to be an associative memory or a learning machine [1][2].

To understand the mechanisms of the human brain we first have to understand the basic unit of the brain. Then we must know how it works in processing the information. It is known that the basic nerve cell or computing unit for biological information processing in the human brain is the neuron. It has been discovered that following the application of a stimulus greater than a threshold value, a pulse of electric potential is generated across the membrane of a neuron. This is called an action potential. An excited neuron transmits an action potential and has a positive or excitatory influence on the recipient nerve cells.

A simplified biological neuron is shown in Figure 2.1. The junction point between axon and dendrite is called the synapse [3]. The inputs, which are the action potentials, transmit through synaptic junctions from the axons of adjacent neurons to the neuron dendrites. They are modulated (or weighted) and carried by the dendrites of the neuron to its cell body. The cell body of the neuron, called the soma, sums the modulated input action potentials and compares with its threshold. An activation is performed if sum of the action potentials is beyond that threshold. Accordingly, the active (or excited) neuron fires an action potential through its axon to the dendrites of other neurons.

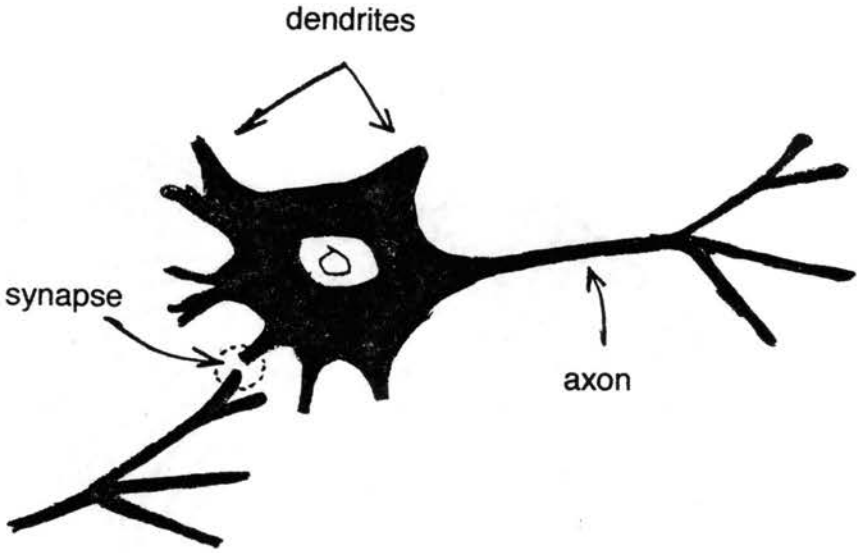


Figure 2.1 A Simplified Biological Neuron

Artificial Neuron

To imitate the biological counterparts an artificial neuron model has been developed (see Figure 2.2). It is also called a single layer perceptron or a network building block. It has inputs (dendrites), connection weights (synapses), a weighted summer that performs the activation function (soma), and output (axon). Also shown in Figure 2.2 is the symbolic representation of the artificial neuron that will be used throughout this paper.

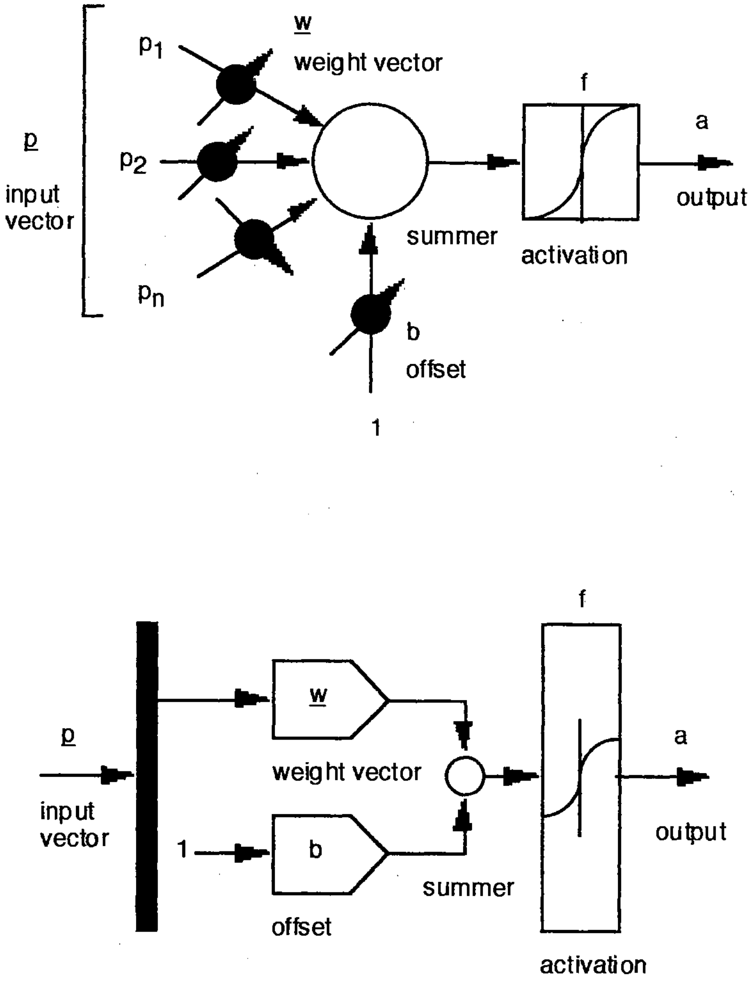
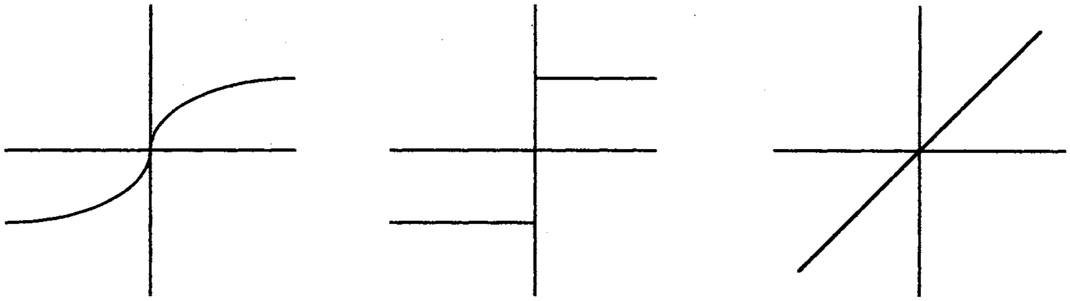


Figure 2.2 Artificial Neuron Model and Its Symbolic Representation

Equation 2.1 is the mathematical operation performed by the artificial neuron. The inner product of the input vector $\underline{p}$ and the weight vector $\underline{w}$ is summed with the offset or threshold b (which has an identity input) and the result is called the net input n. Through the activation function f, the net input n is mapped to the output a.

$$\begin{aligned} n &= \underline{w}^T \underline{p} + b \\ a &= f(n) \end{aligned} \tag{2.1}$$

Typical activation functions are sigmoid, threshold or linear in shape as shown in Figure 2.3.



$a = \dfrac{1}{1+e^{-n}} + c$ $\qquad$ $a = \text{sign}(n)$ $\qquad$ $a = n$

Figure 2.3 Typical Activation Functions

Multilayer Neural Networks

Based on the neuron model just described, the next step is the development of multilayer neural networks. The most commonly used architecture is called the feedforward multilayer network. This network has three components, as shown in Figure 2.4, which is a multi-input multi-output (MIMO) two hidden layer network. The first component is a group of input nodes. The input nodes are not neurons, and thus do not perform any mathematical operation. The one and only function of each input node is to distribute the input to the neurons of the first hidden layer. The second component of the network is the hidden layer(s). A network can have as many hidden layers as the design requires. In multi-hidden layer networks, the output of one hidden layer is the input of the following hidden layer. The last component of the feedforward multilayer network is the output layer which, like the hidden layer, is composed of neurons. The activation function of each neuron can be the same throughout the network or it can be varied by each layer. For all of the networks that are described in this paper, we always use a sigmoid activation function for hidden layers and a linear function for the output layer.

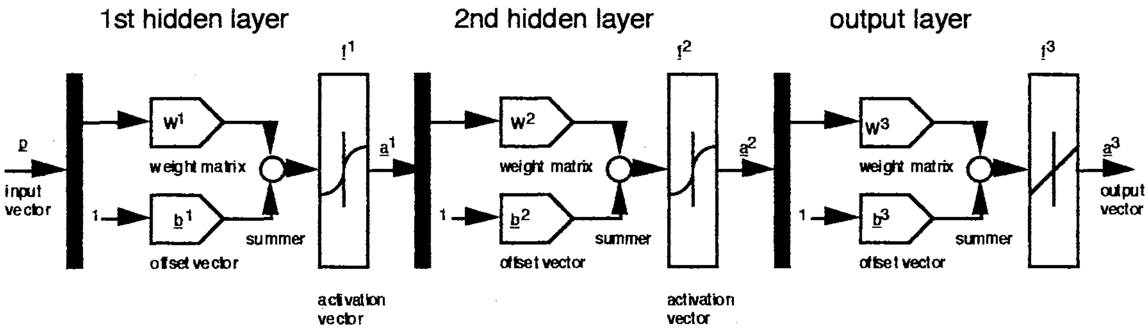


Figure 2.4 A MIMO Two Hidden Layer Network

The mathematical operation of a multilayer network, which is basically derived from the mathematical operation of the artificial neuron, is governed by the following equation (a MIMO two hidden layer network example)

$$\underline{a}^3 = \underline{f}^3(W^3[\underline{f}^2(W^2[\underline{f}^1(W^1\underline{p} + \underline{b}^1)] + \underline{b}^2)] + \underline{b}^3), \tag{2.2}$$

where $\underline{p}$ is the input vector, W^1, W^2, and W^3 are the weight matrices , $\underline{b}^1$, $\underline{b}^2$, and $\underline{b}^3$ are the offset vectors, $\underline{f}^1$, $\underline{f}^2$, and $\underline{f}^3$ are the activation function vectors for each layer, and the output vector is $\underline{a}^3$. The general expression for Equation 2.2 is written

$$\begin{aligned}\underline{a}^M = \underline{f}^M(W^M[\underline{f}^{M-1}(W^{M-1}\cdots \\ [\underline{f}^i(W^i\cdots[\underline{f}^1(W^1\underline{p} + \underline{b}^1)]+\cdots\underline{b}^i)]+\cdots\underline{b}^M)'\end{aligned} \tag{2.3}$$

where $\underline{a}^M$ is the output vector and M is the total number of layers in the network (including all the hidden layers and the output layer).

Another type of multilayer network commonly used in this paper is called a recurrent network, which is derived from the feedforward network. Its layout is shown in Figure 2.5. The most significant component added is called the tapped delay line. This component feedbacks the time-lagged output to the network input. This network will be discussed in more detail in Chapter 5.

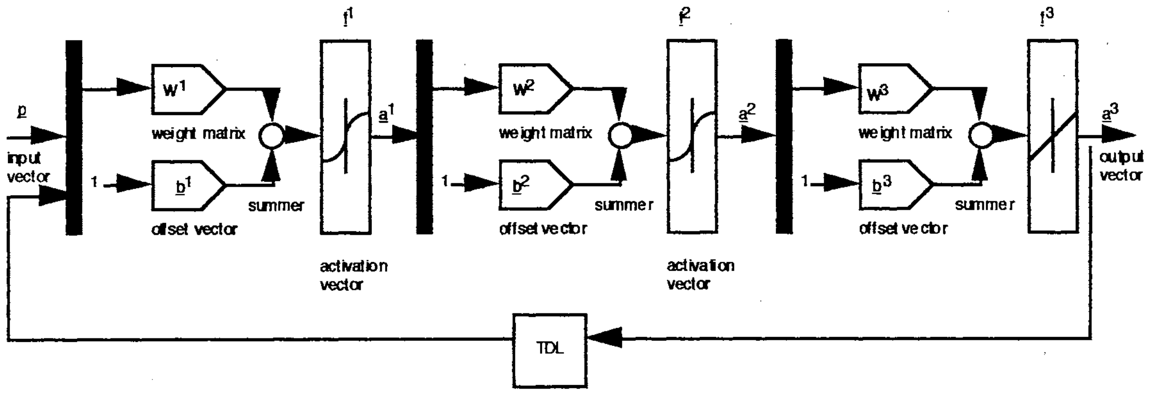


Figure 2.5 Recurrent Multilayer Network

Function Approximation

Several methods, like polynomials, trigonometric series, orthogonal functions, and splines have all been used for function approximation. Evolving in recent years, neural networks have been demonstrated as an additional tool for function approximation. In fact, one of the reasons neural networks have been found to have so many successes in a wide range of applications, such as pattern recognition, signal processing and, of course, control systems, is due to the capability of the network to perform function approximation.

Under classic approximation theory, the Weierstrass theorem [4] provided a rigorous proof that any arbitrary continuous function on a compact set can be approximated to any degree of accuracy by polynomials as well as other approximation schemes. Based on this Weierstrass theorem, Hornik et al. [5] proved that a layered neural network is a nonlinear parametric model and can approximate any continuous input-output relation.

Even with the proof, still a major concern remains. Does a systematic method exist for determining the number of layers/neurons necessary to achieve a desired degree of accuracy for the function being approximated? So far, the answer is no. In the absence of this systematic method, several tests have been performed in the next section which compare the capabilities of different network architectures. These tests provide an attempt to account for some practical considerations in choosing the number of layers and the number of neurons during the design of neural network architectures.

Network Performance and Network Architecture

To compare the performances among different network architectures, we start by examining a single hidden layer network. The limitation of the network function approximating ability is tested as the complexity of the function to be approximated (the sample function) increases. Then, still using a single hidden layer network, a second test is performed by increasing the number of neurons in the hidden layer. In this test, the complexity of the sample function is fixed. The third test, based on having the same number of adjustable parameters (weights and offsets) in each network, compared the single hidden layer network (as in test 1) with a two hidden layer network. In the last test, several pairs of networks that have different numbers of hidden layers, but the same total number of adjustable parameters, were compared.

A single-input single-output (SISO) feedforward multilayer network is the common architecture used for all of the tests. The general expression for the architecture is $1\text{-}n_1\text{-}n_2\text{-}\dots\ \text{-}n_i\text{-}\dots\ \text{-}n_{M\text{-}1}\text{-}1\_N$, where n_i is the number of neurons in i-th hidden layer, M is the number of layers (thus we have M-1 hidden layers), and N is the total number of adjustable parameters in each network.

The sample function to be approximated in each test is described by

$$t = \sin(\frac{\pi}{2}mp), \qquad (2.4)$$

where m is degree of complexity (which can be any positive integer), and the desired (or target) output t is obtained as a sinusoidal function of the input p. Throughout this document, the way to present training data to the neural network is always in a batch mode. So, an input vector $\underline{p}$ is formed with 401 data points from -1 to 1 with a increasing interval of 0.005. By Equation 2.4, a corresponding target output vector $\underline{t}$ is thereby obtained. The sample functions for m=1 and m=8 are shown in Figure 2.6.

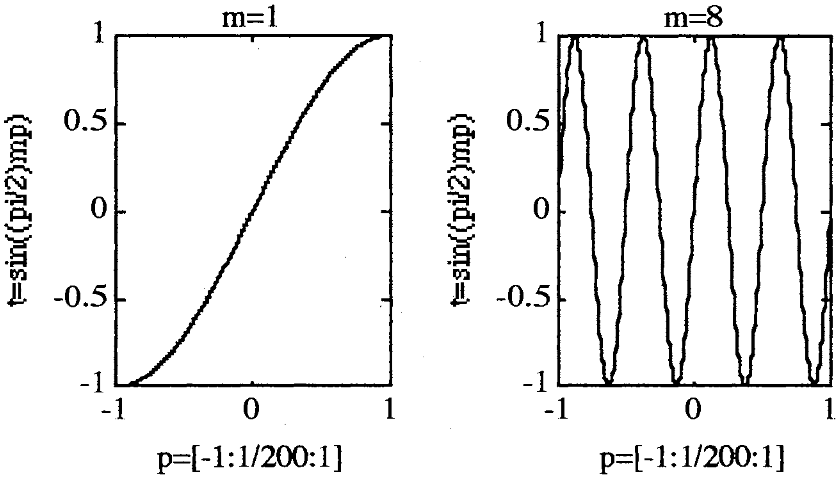


Figure 2.6 The Sample Functions

The learning algorithm is Marquardt optimization with backpropagation. It will be discussed in detail in Chapter 3. The learning process has four steps;

(1) present the input data set $\underline{p}$ to the neural network in a single batch.

(2) calculate the network output $\underline{a}^M$ by using Equation 2.3.

(3) take the sum of squared error between $\underline{a}^M$ and target data $\underline{t}$ to obtain the performance index.

(4) stop the learning process if the performance index reaches the desired value, otherwise adjust the weights of the network with the learning algorithm before repeating the same learning process again.

Steps (1) to (4) defines one learning epoch.

Another unanswered problem in network performance is how to choose the initial weights at the start of training a network. For all of the network trainings described in this document, the initial weights were chosen as random numbers that are uniformly distributed between -1 and 1. It was discovered that, from our empirical experience, different random initial weights sets usually result in different learning outcomes. These outcomes, such as the learning speed and even convergence (whether or not a satisfactory approximation is reached) vary greatly. In order to smooth out those outcomes, each test case is trained with ten trials (which means ten different initial weight sets) in the following tests. Therefore, all of the learning results are obtained as the average of those ten trials.

Test 1 : Function Approximation for the Single Hidden Layer Network

Network : 1-4-1_13

Desired performance index : 0.0401 (401 points)

Maximum learning epochs : 800

Function complexity : m = 1,2,..,8

TABLE 2.1 Results of Test 1

Case #	Function Complexity m	# Successes in Ten Trials	Average CPU Time on Centris 650	Average # of Epochs
1	1	10/10	9.14	6.7
2	2	10/10	29.00	20.3
3	3	10/10	77.57	54.4
4	4	10/10	109.60	74.5
5	5	10/10	1158.16	800.0
6	6	2/10	982.51	668.7
7	7	6/10	1132.84	800.0
8	8	0/10	1183.37	800.0

The results from case 1 to case 5 in Table 2.1 were as expected. That is, as the complexity of the sample function increased, longer learning CPU time was required. However, in case 6, the network performance is reaching its limit with only two satisfactory approximations obtained out of total of ten trials. It is not clear, at this point, why the success rate of case 7 is higher than case 6. The last case, with no successful approximation, has a function complexity which is beyond the performance capability of the network.

Test 2 : The Oversized Networks

Network : 1-n-1_N where n=5,6,..,12

Desired performance index : 0.0401

Maximum learning epochs : 800

Function complexity : m = 8

TABLE 2.2 Results of test 2

Case #	Network Structure	# Successes in Ten Trials	Average CPU Time on Centris 650	Average # of Epochs
1	1-5-1_16	2/10	1378.86	800.0
2	1-6-1_19	9/10	709.87	355.3
3	1-7-1_22	10/10	501.38	221.7
4	1-8-1_25	10/10	566.68	219.9
5	1-9-1_28	10/10	441.98	151.0
6	1-10-1_31	10/10	531.00	164.8
7	1-11-1_34	10/10	562.18	158.9
8	1-12-1_37	10/10	582.50	148.9

The most difficult approximation task performed in test 1 is in the last case, which has function complexity of eight. For all of the cases in test 2, the function complexity is fixed at eight, but the number of neurons in the hidden layer is increased from one case to the next. The first fully successful rate of approximation is reached when n is 7, in case 3. In terms of the lowest learning CPU time, network 1-9-1_28 is the best. Comparing case 3 to case 4 to 8, we observed that oversized networks may take fewer epochs to converge, but they take more CPU time to learn and require more memory.

Test 3 : Function Approximation for the Two Hidden Layer Network

Network : 1-2-2-1_13
Desired performance index : 0.0401
Maximum learning epochs : 800
Function complexity : m = 1,2,..,8

TABLE 2.3 Results of Test 3

Case #	Function Complexity m	# Successes in Ten Trials	Average CPU Time on Centris 650	Average # of Epochs
1	1	10/10	25.06	15.8
2	2	9/10	191.30	116.9
3	3	10/10	439.21	267.1
4	4	3/10	991.73	603.7
5	5	2/10	1302.40	800.0
6	6	0/10	1294.27	800.0
7	7	0/10	1298.31	800.0
8	8	0/10	1301.20	800.0

The two hidden layer network tested here and the single hidden layer network used in Test 1 have the same total number of adjustable parameters. After comparing the results of this test and Test 1, it can be concluded that the single hidden layer network performed better than the two hidden layer network, not only in terms of learning speed, but also the rate of successful approximations.

Test 4 : Comparisons between Single Hidden Layer Network and Multi-hidden Layer Network

Network pairs : [1-7-1_22 and 1-3-3-1_22]

[1-8-1_25 and 1-2-2-2-2-1_25]
[1-11-1_34 and 1-3-3-3-1_34]

Desired performance index : 0.0401

Maximum learning epochs : 800

Function complexity : m = 2

TABLE 2.4 Results of Test 4

Case #	Network Structure	# Successes in Ten Trials	Average CPU Time on Centris 650	Average # of Epochs
1	1-7-1_22	10/10	32.48	14.7
2	1-3-3-1_22	10/10	46.10	20.3
3	1-8-1_25	10/10	38.13	15.5
4	1-2-2-2-2-1_25	9/10	429.13	131.1
5	1-11-1_34	10/10	41.31	12.5
6	1-3-3-3-1_34	10/10	85.76	23.1

The immediate conclusion one makes from Test 4 is that single hidden layer networks often perform better in terms of learning speed. With increasing number of adjustable parameters, more training time is needed. In case 4, the four hidden layer network not only took the most CPU time, but also failed to converge (reach satisfactory approximation) once in ten trials.

Summary

In this chapter, the basic concepts of neural networks were described, and the function approximation capability of the networks was demonstrated through a series of tests. The neural network learning process, which has been introduced in this chapter, is very important and essential in neurocontrol. That is why we will dedicate the next chapter to its discussion.

CHAPTER III

BACKPROPAGATION AND OPTIMIZATION

After describing neural networks in Chapter 2, in this chapter we will present the process of training neural networks to perform function approximation. The chapter begins with a description of the supervised learning process. Then, the backpropagation method is described, followed by discussions of three optimization techniques. The conclusion of this chapter focuses on implementing the neural network learning process.

The Supervised Learning Process

The type of learning process discussed in this chapter and throughout the document is called supervised learning. Other known learning types, like unsupervised learning and reinforcement learning, are outside the scope of this discussion. Supervised learning applies to a situation in which a neural network functions as a replacement for an input-output mapping relation. To achieve that, a set of desired input-output pairs (p_1,t_1), (p_2,t_2), ... (p_j,t_j), ... , which is derived from the mapping relation, is supplied for training a neural network.

The learning process is made up of a sequence of learning iterations called epochs or sweeps. Each learning epoch starts by presenting an input vector $\underline{p}_i$ from the desired input-output pair ($\underline{p}_i$, $\underline{t}_i$), to the neural network. Then, a forward computation is performed resulting in the network output $\underline{a}_i$. (where subscript i corresponds to the i-th learning epoch). The error vector

$$\underline{e}_i = \underline{t}_i - \underline{a}_i \qquad (3.1)$$

is obtained after comparing the network output $\underline{a}_i$ with the desired output $\underline{t}_i$. The objective function F($\underline{x}$), which is commonly defined as the sum of squared error,

$$F(\underline{x}) = (\underline{t}_i - \underline{a}_i)^T (\underline{t}_i - \underline{a}_i), \qquad (3.2)$$

is set up as a performance criterion. It will be used to adjust the current parameters of the network in order to produce a better approximation.

There are two main tasks in each learning epoch, after the objective function is obtained: 1) By backpropagation, calculate the partial derivatives of the objective function with respect to the parameter vector. 2) By some optimization technique, determine the search direction towards the global minimum of the error surface in the parameter hyperspace. At the end of each learning epoch, the parameter vector $\underline{x}$ is moved along the search direction. The newly updated parameter vector will be used at the start of the next learning epoch. As for the very first learning epoch, the process starts with a predetermined initial parameter vector. The learning iterations stop when the objective function, which is a index of the degree of accuracy, reaches some desired level. This defines the supervised learning process.

Backpropagation

The backpropagation method was originally introduced by Paul Werbos [6], but it was not established as a mainstay in neurocomputing until the work done by David Rumelhart et al [7]. Backpropagation has since become an essential tool in supervised learning. It is essential because the most efficient way to calculate the first derivatives of the objective function with respect to the adjustable parameters of the network is through backpropagation. With these derivatives, we can use the optimization techniques which will be described in the next section to minimize the objective function in training the neural network.

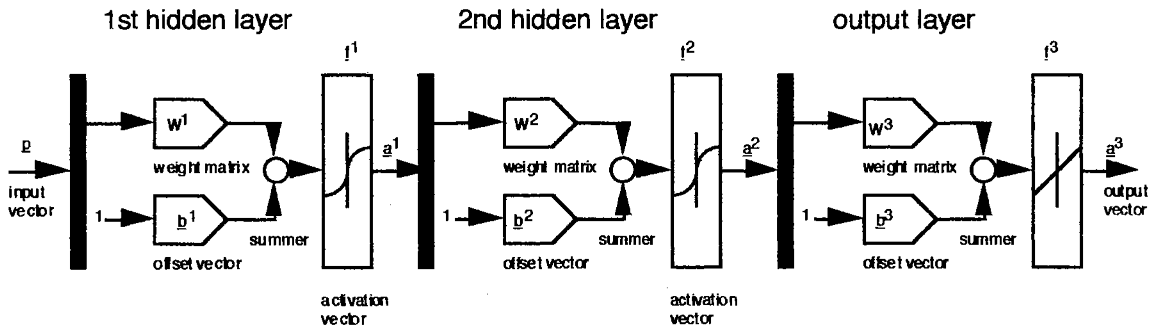


Figure 3.1 A MIMO Two Hidden Layer Network

In a multilayer neural network, as shown in Figure 3.1, the general mathematical operation performed by the k-th layer neurons is

$$\underline{n}^k = W^k \underline{a}^{k-1} + \underline{b}^k$$
$$\underline{a}^k = \underline{f}^k(\underline{n}^k) \qquad (3.3)$$
$$\text{where } k = 1,2,\ldots M$$

where W^k is the weight matrix, $\underline{b}^k$ is the offset vector, $\underline{f}^k$ is the activation function, $\underline{n}^k$ is the net input and $\underline{a}^k$ is the output (or the activation) of the k-th layer of the network. The activations for k=0 (the input nodes) and k=M (the output layer) are, respectively,

$$\underline{a}^0 = \underline{p}$$
$$\text{and} \qquad (3.4)$$
$$\underline{a} = \underline{a}^M,$$

where $\underline{p}$ is the input vector and $\underline{a}$ is the output vector. The parameter vector $\underline{x}$ contains all the elements of the weight matrices and the offset vectors. The objective function $F(\underline{x})$, which is defined in Equation 3.2, cannot be obtained until the network output is computed. That means that a forward computation from the first layer to the output layer must be performed first. For the same reason, the derivatives with respect to the network parameters cannot be calculated until the objective function of the learning epoch is available. Once the objective function is obtained, it is logical, as we will explain later in this section, to calculate the derivatives from the output layer backward to the first layer. This method is thus called backpropagation.

The partial derivative of the objective function F($\underline{x}$) with respect to the assemble of weights and offsets of the k-th layer $\underline{w}^k$ is

$$\frac{\partial F(\underline{x})}{\partial \underline{w}^k} = \left(\frac{\partial F(\underline{x})}{\partial \underline{n}^k}\right)^T \frac{\partial \underline{n}^k}{\partial \underline{w}^k} \tag{3.5}$$

The sensitivity of the k-th layer is defined

$$\underline{\delta}^k = \frac{\partial F(\underline{x})}{\partial \underline{n}^k} \tag{3.6}$$

Suppose we want to compute this sensitivity, then

$$\begin{aligned} \underline{\delta}^k &= \frac{\partial F(\underline{x})}{\partial \underline{n}^k} \\ &= \left(\frac{\partial F(\underline{x})}{\partial \underline{n}^{k+1}}\right)^T \frac{\partial \underline{n}^{k+1}}{\partial \underline{n}^k} \\ &= \left(\frac{\partial F(\underline{x})}{\partial \underline{n}^M}\right)^T \left(\frac{\partial \underline{n}^M}{\partial \underline{n}^{M-1}}\right)^T \cdots \frac{\partial \underline{n}^{k+1}}{\partial \underline{n}^k} \end{aligned} \tag{3.7}$$

We cannot make this computation until we have the sensitivity for the last (output) layer $\underline{\delta}^M$

$$\begin{aligned} \underline{\delta}^M &= \frac{\partial F(\underline{x})}{\partial \underline{n}^M} \\ &= \frac{\partial[\frac{1}{2}(\underline{t}-\underline{a})^T(\underline{t}-\underline{a})]}{\partial \underline{n}^M} \\ &= -(\underline{t}-\underline{a})\frac{\partial \underline{a}}{\partial \underline{n}^M} \\ &= -(\underline{t}-\underline{a})(\underline{f}^M_{\underline{n}^M})' \end{aligned} \tag{3.8}$$

where the term $(\underline{f}^M_{\underline{n}^M})'$ in the above equation stands for the first derivative of the output layer activation function $\underline{f}^M$ while the net input is $\underline{n}^M$. By investigating Equations 3.7 and 3.8 further, we find that there is a recurrent relation between $\underline{\delta}^k$ and $\underline{\delta}^{k+1}$ such that

$$\begin{aligned}
\underline{\delta}^{k} &= \frac{\partial F(\underline{x})}{\partial \underline{n}^{k}} \\
&= \left(\frac{\partial \underline{n}^{k+1}}{\partial \underline{n}^{k}}\right)^{T} \frac{\partial F(\underline{x})}{\partial \underline{n}^{k+1}} \\
&= \left(\frac{\partial \underline{n}^{k+1}}{\partial \underline{n}^{k}}\right)^{T} \underline{\delta}^{k+1} \qquad (3.9) \\
&= [(\underline{f}^{k}_{\underline{n}^{k}})' \, (W^{k+1})^{T}]\underline{\delta}^{k+1}
\end{aligned}$$

where $k = 1, 2, \cdots M - 1$

From Equations 3.7 through 3.9 , we can now understand now why it is logical to calculate derivatives from the last layer backwards to the first layer.

Optimization Techniques [8]

In order to train a neural network to be a function approximator, we need to optimize (or minimize) the objective function $F(\underline{x})$, which is usually defined as the sum of squared error. This explains why we need optimization techniques for training networks. The standard optimization algorithm has the form

$$\underline{x}_{i+1} = \underline{x}_{i} + \alpha_{i} \underline{s}_{i} , \qquad (3.10)$$

where $\underline{x}_i$ is the parameter vector at epoch i, $\underline{s}_i$ is the search direction, and α_i is a scalar called the learning rate or step size. The search direction vector $\underline{s}_i$ is obtained from the backpropagation method and the optimization technique. There are many optimization techniques. It is the computation of $\underline{s_i}$ that distinguishes one optimization method from another. In this section we will discuss three methods: the steepest descent method, the Marquardt method and the conjugate directions method. These methods all use only the first derivatives of the objective function to determine the search direction vector $\underline{s}_i$.

Steepest Descent Method

The function of the search direction vector is to decrease the objective function at each learning iteration

$$F(\underline{x}_{i+1}) < F(\underline{x}_i) \qquad (3.11)$$

To achieve that decrease, consider the following Taylor series expansion

$$\begin{aligned} F(\underline{x}_{i+1}) &= F(\underline{x}_i + \alpha_i \underline{s}_i) \\ &\approx F(\underline{x}_i) + \alpha_i \nabla F(\underline{x}_i) \underline{s}_i{}' \end{aligned} \qquad (3.12)$$

where $\nabla F(\underline{x}_i)$ is the gradient of the objective function at epoch i. For a positive learning rate α_i, we must have

$$\nabla F(\underline{x}_i) \underline{s}_i < 0. \qquad (3.13)$$

This is called a descent direction. Equation 3.12 can be rewritten as

$$F(\underline{x}_{i+1}) \approx F(\underline{x}_i) + \alpha_i \|\nabla F(\underline{x}_i)\| \, \|\underline{s}_i\| \cos\theta \qquad (3.14)$$

Where $\|\cdot\|$ represents norm, and θ is the angle between $\nabla F(\underline{x}_i)$ and $\underline{s}_i$. If $\|\nabla F(\underline{x}_i)\|$ and $\|\underline{s}_i\|$ are fixed, then with a variable θ, the greatest reduction in Equation 3.14 is obtained when $\theta = \pi$. Thereby the steepest descent is defined

$$\underline{s}_i = -\nabla F(\underline{x}_i) \qquad (3.15)$$

and we have the steepest descent method

$$\underline{x}_{i+1} = \underline{x}_i - \alpha_i \nabla F(\underline{x}_i) \qquad (3.16)$$

If the learning rate α_i is fixed, then it has to be very small to ensure that convergence occurs in the learning process. This is explained in Figure 3.2 where an abstract error surface is presented. Several contour lines are drawn on the abstract error surface. Each contour line represents the points that have the same value for the objective function F. The real minimum $\underline{x}_*$ is located between the two innermost contour lines, which have the two lowest values of F among the other contour lines. We assume that after some iterations, the estimated minimum $\underline{x}_i$ is closing toward the true minimum $\underline{x}_*$. If the step size is too large then an oscillation occurs between the two innermost contour lines. The estimated minimum tries to settle down at the true minimum, however, with the large step size the estimated minimum tends to oscillate. On the other hand, if the learning rate is very small, the oscillation may not occur, but more steps (longer learning time) are required to reach the true minimum.

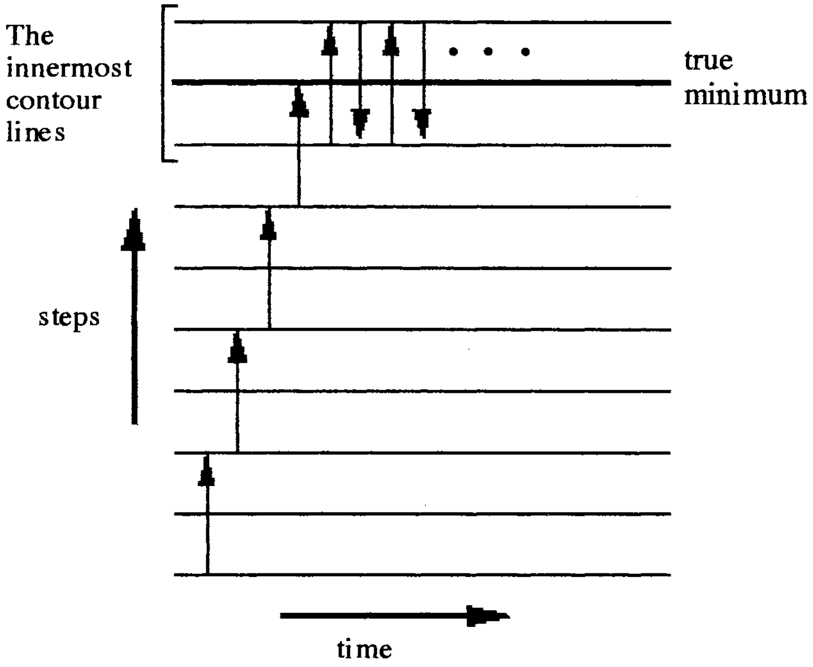


Figure 3.2 An Illustration of Fixed Step Size Learning

An improvement can be made if some ad hoc techniques are used to vary (or to adapt) the step size in each learning epoch. As shown in Figure 3.3, the main idea is simple. We would like the step size to be large at the start of the learning process and to decrease during the process. With the larger step size at the beginning, the estimate can move faster towards the minimum from a distant starting point. Then, with decreasing step size, the oscillation caused by a too large step size can be prevented. Therefore, with the variation in step size, convergence is ensured. This is called a variable (or adaptive) learning rate algorithm.

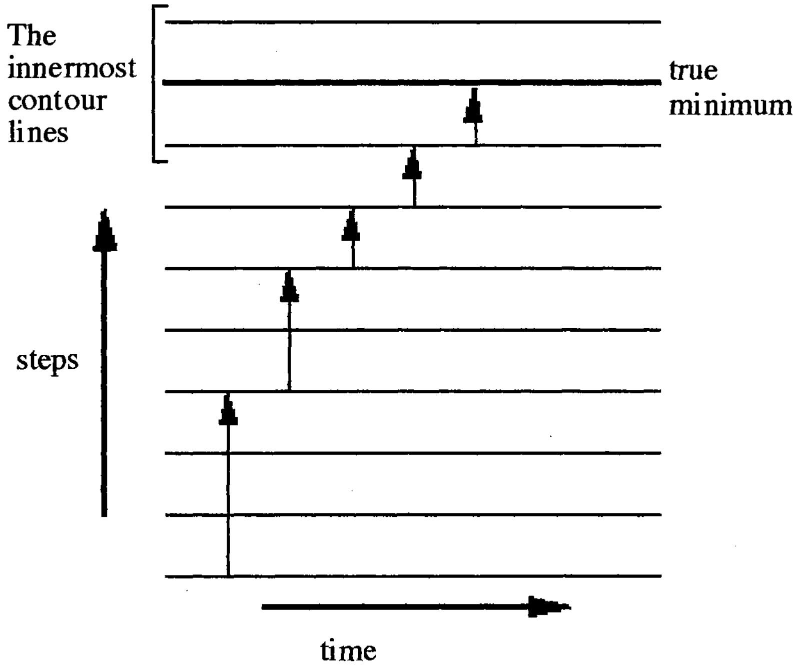


Figure 3.3 An Illustration of the Adaptive Learning Rate Algorithm

Marquardt Method

The Marquardt method was derived from the Newton method. An objective function can be approximated by a quadratic function with positive definite Hessian matrix in the immediate neighborhood of a strong minimum. The Newton method was developed using this property. Consider the following second order Taylor series expansion of the objective function about the estimated minimum point $\underline{x}_i$

$$\begin{aligned} F(\underline{x}_{i+1}) &= F(\underline{x}_i + \Delta\underline{x}_i) \\ &\approx F(\underline{x}_i) + \nabla F(\underline{x}_i)^T \Delta\underline{x}_i + \frac{1}{2}\Delta\underline{x}_i^T \nabla^2 F(\underline{x}_i)\Delta\underline{x}_i \end{aligned} \tag{3.17}$$

As we know, for a second order equation, the minimum (or maximum) is located at the point with zero first derivative. So we take the gradient of Equation 3.17 with respect to $\Delta\underline{x}_i$ and set it to zero

$$\nabla F(\underline{x}_i) + \nabla^2 F(\underline{x}_i)\Delta\underline{x}_i = 0 \tag{3.18}$$

The minimum $\underline{x}_*$ can then be reached with one weight update

$$\Delta\underline{x}_i = -\nabla^2 F(\underline{x}_i)^{-1} \nabla F(\underline{x}_i) \tag{3.19}$$

To state the Newton method

$$\underline{x}_{i+1} = \underline{x}_i - \nabla^2 F(\underline{x}_i)^{-1} \nabla F(\underline{x}_i) \tag{3.20}$$

The advantage of the Newton method is that it generally converges in fewer learning epochs than steepest descent. However, the disadvantage of the Newton method is its need for second derivatives, which requires a lot of computations. The following modification, called the Gauss-Newton algorithm, avoids this calculation requirement by approximating the second derivatives with first derivatives. Consider the objective function that is defined as the sum of squares of other functions

$$\begin{aligned} F(\underline{x}) &= f_1^2(\underline{x}) + f_2^2(\underline{x}) + \cdots + f_N^2(\underline{x}) \\ &= [f(\underline{x})]^T [f(\underline{x})] \end{aligned} \tag{3.21}$$

where $\underline{f}(\underline{x}) = [f_1(\underline{x})\ f_2(\underline{x}) \cdots f_N(\underline{x})]^T$

We then take the gradient of Equation 3.21 with respect to $\underline{x}$ (The size of $\underline{x}$ is n)

$$\nabla F(\underline{x}) = 2[J(\underline{x})]^T [\underline{f}(\underline{x})]$$

$$\text{where } J(\underline{x}) = \begin{bmatrix} \frac{\partial f_1(\underline{x})}{\partial w_1} & \frac{\partial f_1(\underline{x})}{\partial w_2} & \cdots & \frac{\partial f_1(\underline{x})}{\partial w_n} \\ \frac{\partial f_2(\underline{x})}{\partial w_1} & \frac{\partial f_2(\underline{x})}{\partial w_2} & \cdots & \frac{\partial f_2(\underline{x})}{\partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_N(\underline{x})}{\partial w_1} & \frac{\partial f_N(\underline{x})}{\partial w_2} & \cdots & \frac{\partial f_N(\underline{x})}{\partial w_n} \end{bmatrix} \tag{3.22}$$

The second derivative of Equation 3.21 is

$$\nabla^2 F(\underline{x}) = 2[J(\underline{x})]^T [J(\underline{x})] + 2\sum_{i=1}^{N} f_i(\underline{x}) \nabla^2 f_i(\underline{x}) \tag{3.23}$$

By ignoring the second term in the above equation, we have

$$\nabla^2 F(\underline{x}) \approx 2[J(\underline{x})]^T [J(\underline{x})] \tag{3.24}$$

After replacing the terms of $\nabla^2 F$ and ∇F in Equation 3.20 (the Newton method) with Equation 3.22 and 3.24 respectively, we have the Gauss-Newton algorithm

$$\underline{x}_{i+1} = \underline{x}_i - [J(\underline{x})^T J(\underline{x})]^{-1} [J(\underline{x})^T \underline{f}(\underline{x})] \tag{3.25}$$

If the term $J(\underline{x})^T J(\underline{x})$ in Equations 3.25 is not positive definite, and the inverse term does not exist, then we have a problem in implementing the Gauss-Newton algorithm. To overcome this problem, Levenberg added an additional term, an identity matrix I times a positive μ, to the term $J(\underline{x})^T J(\underline{x})$ of the Gauss-Newton algorithm

$$\underline{x}_{i+1} = \underline{x}_i - [\nabla^2 F(\underline{x}_i) + \mu I]^{-1} \nabla F(\underline{x}_i). \tag{3.26}$$

Assume that the eigenvalues and eigenvectors of the Hessian matrix $\nabla^2 F(\underline{x})$ in the above equation are $(\lambda_1, \lambda_2, \cdots, \lambda_N)$ and $(\underline{\Delta}_1, \underline{\Delta}_2, \cdots, \underline{\Delta}_N)$ respectively, then

$$\begin{aligned}[\nabla^2 F(\underline{x}) + \mu I]\underline{\Delta}_j &= \nabla^2 F(\underline{x})\underline{\Delta}_j + \mu I \underline{\Delta}_j \\ &= \lambda_j \underline{\Delta}_j + \mu \underline{\Delta}_j \\ &= (\lambda_j + \mu)\underline{\Delta}_j\end{aligned} \tag{3.27}$$

Thus the new eigenvalues and eigenvectors of the modified Hessian matrix in Equation 3.27 are $(\lambda_1+\mu, \lambda_2+\mu, \cdots, \lambda_N+\mu)$ and $(\underline{\Delta}_1, \underline{\Delta}_2, \cdots, \underline{\Delta}_N)$ respectively. To obtain a positive definite Hessian matrix, one can always increase the value of μ until each new eigenvalue is positive.

The Levenberg algorithm is a complimentary method between the Gauss-Newton method, when μ_i is small, and the steepest descent algorithm, when μ_i is large. It is complimentary because the Levenberg algorithm takes the advantages of fast learning speed from the Gauss-Newton method and exact direction learning from the steepest descent algorithm. Based on this observation, Marquardt suggested that one can start with a small number for μ_i at the beginning of the training. Then we increase μ_i by a factor β until the objective function is decreased, as described in Equation 3.11. This is assured because increasing μ_i eventually is equivalent to taking a small step in the steepest descent direction. To avoid the problem of having the value of μ_i get big, the same factor, β, can be used to decrease μ_i at the end of each learning epoch.

Conjugate Directions Method

Like the Marquardt algorithm, the conjugate directions method is another way to perform optimization without the need for second derivatives. Consider a quadratic function which is in the form of

$$F(\underline{x}) = c + \underline{g}^T \underline{x} + \frac{1}{2} \underline{x}^T H \underline{x}. \tag{3.28}$$

Its gradient and Hessian matrix are, respectively,

$$\begin{aligned} &\nabla F(\underline{x}) = H\underline{x} + \underline{g} \\ &\text{and} \\ &\nabla^2 F(\underline{x}) = H. \end{aligned} \tag{3.29}$$

Then, a set of vectors $\underline{s}_i$ is said to be mutually conjugate with respect to the Hessian matrix H if and only if

$$\underline{s}_i^T H \underline{s}_j = 0 \qquad i \neq j \tag{3.30}$$

As we know, if the Hessian matrix is symmetric then its eigenvectors are orthogonal. In this case, the set of the eigenvectors is a set of mutually conjugate vectors

$$\begin{aligned} \underline{\Delta}_i^T H \underline{\Delta}_j &= \underline{\Delta}_i^T (\lambda_j \underline{\Delta}_j) \\ &= \lambda_j (\underline{\Delta}_i^T \underline{\Delta}_j) \\ &= 0 \end{aligned} \tag{3.31}$$

Let $\underline{\Delta}_i$ be the search direction vector $\underline{s}_i$. If the Hessian matrix in Equation 3.31 is positive definite, then it can be shown that the exact minimum of a quadratic function will be reached in a maximum of n steps, using

$$\Delta\underline{x}_i = \underline{x}_{i+1} - \underline{x}_i = \alpha_i \underline{s}_i \qquad (3.32)$$

where n is the dimension of $\underline{x}$ and α_i is the exact single step needed to reach the minimum along $\underline{s}_i$. To eliminate the need for a Hessian matrix (second derivatives) in Equation 3.31, we then combine Equations 3.28 and 3.29

$$\begin{aligned} \Delta[\nabla F(\underline{x}_i)] &= \nabla F(\underline{x}_{i+1}) - \nabla F(\underline{x}_i) \\ &= H(\underline{x}_{i+1} - \underline{x}_i) \\ &= H[\Delta\underline{x}_i] \end{aligned} \qquad (3.33)$$

The conjugate condition from the i-th learning epoch to the (i+1)-th learning epoch can be found in Equation 3.30. If we multiply both sides of the equation by α_i, and then combine it with Equations 3.32 and 3.33 we obtain

$$\begin{aligned} [\alpha_i \underline{s}_i]^T H\underline{s}_{i+1} &= \Delta\underline{w}_i^T H\underline{s}_{i+1} \\ &= \Delta[\nabla F(\underline{x}_i)]^T \underline{s}_{i+1} \\ &= 0. \end{aligned} \qquad (3.34)$$

Thus the ground for computing the next search direction vector $\underline{s}_{i+1}$ is established. Numerous solutions can be found to satisfy Equation 3.34. One set of directions which satisfy Equation 3.34 are

$$\begin{aligned} &\underline{s}_1 = \nabla F(\underline{x}_1) \\ &\underline{s}_{i+1} = \nabla F(\underline{x}_{i+1}) + \beta_{i+1}\underline{s}_i \quad i = 1,2,\cdots n-1 \\ &\beta_{i+1} = [\nabla F(\underline{x}_{i+1})]^T[\nabla F(\underline{x}_{i+1})] / [\nabla F(\underline{x}_i)]^T[\nabla F(\underline{x}_i)] \end{aligned} \qquad (3.35)$$

where n is the dimension of the parameter vector $\underline{x}$.

Implementation

The purpose of this section is to offer a series of equations for implementing the neural network learning process from a programming point of view. Those equations, which are organized in the order of real events, are summarized from the previous sections about backpropagation and optimization techniques. As discussed earlier in this chapter, the learning process starts with a forward computation. The forward computation obtains not only the network output, but also the net inputs and the outputs (activations) for each layer in the network. Then, with these net inputs and outputs, a backpropagation is performed to find the first derivatives of the objective function with respect to each adjustable parameter of the network. Those derivatives are used in the last stage of the learning process: the optimization. The function of the optimization procedure is to calculate the change to be made in each parameter. Once the parameters are adjusted accordingly, the process is repeated until the objective function reaches the goal.

Forward Computation

Input nodes :

$$\underline{a}^0 = \underline{p} \tag{3.36}$$

The k-th hidden layer :

$$\begin{aligned} \underline{n}^k &= W^k \underline{a}^{k-1} + \underline{b}^k \\ \underline{a}^k &= \underline{f}^k(\underline{n}^k) \quad \text{where } k = 1,2,\cdots M-1. \end{aligned} \tag{3.37}$$

The output layer :

$$\begin{aligned} \underline{n}^{M} &= W^{M}\underline{a}^{M-1} + \underline{b}^{M}, \\ \underline{a}^{M} &= \underline{f}^{M}(\underline{n}^{M}). \end{aligned} \tag{3.38}$$

Backpropagation

The output layer :

$$\begin{aligned} \underline{\delta}^{M} &= -(\underline{t} - \underline{a})(\underline{f}^{k}_{\underline{n}^{k}})', \\ \nabla F(W^{M}) &= \underline{\delta}^{M}\underline{a}^{M-1}, \\ \nabla F(\underline{b}^{M}) &= \underline{\delta}^{M}. \end{aligned} \tag{3.39}$$

The k-th hidden layer :

$$\begin{aligned} \underline{\delta}^{k} &= (\underline{f}^{k}_{\underline{n}^{k}})'[W^{k+1}]^{T}\underline{\delta}^{k+1}, \\ \nabla F(W^{k}) &= \underline{\delta}^{k}\underline{a}^{k-1}, \\ \nabla F(\underline{b}^{k}) &= \underline{\delta}^{k} \text{ where } k = M-1, M-2, \cdots 3, 2. \end{aligned} \tag{3.40}$$

The first hidden layer :

$$\begin{aligned} \underline{\delta}^{1} &= (\underline{f}^{1}_{\underline{n}^{1}})'[W^{2}]^{T}\underline{\delta}^{2}, \\ \nabla F(W^{1}) &= \underline{\delta}^{1}\underline{p}, \\ \nabla F(\underline{b}^{1}) &= \underline{\delta}^{1}. \end{aligned} \tag{3.41}$$

Optimization

For simplicity, all of the weights and offsets of the network in the i-th learning epoch are lumped into the following single parameter vector

$$x_i = [(\underline{w}^1)^T (\underline{w}^2)^T \cdots (\underline{w}^k)^T \cdots (\underline{w}^M)^T]^T \qquad (3.42)$$

where $\underline{w}^k$ is the ensemble of the weights and offsets of the k-th layer. To update the parameter vector for each new learning epoch, one can use the general equation

$$\underline{x}_{i+1} = \underline{x}_i + \Delta \underline{x}_i \qquad (3.43)$$

Three optimization techniques discussed in this chapter are

-- The steepest descent method

$$\Delta \underline{x}_i = -\alpha_i \nabla F(\underline{x}_i) \qquad (3.44)$$

-- The Marquardt method

$$\Delta \underline{x}_i = [J^T(\underline{x}_i) J(\underline{x}_i) + \mu I]^{-1} [J^T(\underline{x}_i)(\underline{t} - \underline{a})] \qquad (3.45)$$

-- The Conjugate directions method

(1) The first search direction

$$\begin{aligned} \underline{s}_1 &= -\nabla F(\underline{x}_1) \\ \Delta \underline{x}_1 &= \alpha_1 \underline{s}_1 \end{aligned} \qquad (3.46)$$

(2) The hereafter search directions

$$\begin{aligned} \beta_i &= ([\nabla F(\underline{x}_i)]^T [\nabla F(\underline{x}_i)]) / ([\nabla F(\underline{x}_{i-1})]^T [\nabla F(\underline{x}_{i-1})]) \\ \underline{s}_i &= -\nabla F(\underline{x}_i) + \beta_i \underline{s}_{i-1} \\ \Delta \underline{x}_i &= \alpha_i \underline{s}_i \quad \text{where } i = 2, 3, \cdots, n-1 \end{aligned} \qquad (3.47)$$

Summary

The supervised learning process is thoroughly discussed in this chapter. The essence of the learning process is optimization and backpropagation is a necessary tool in optimization. This necessity of the backpropagation in the learning process is stressed. Three optimization techniques are examined. A comparison of those techniques are contained in the work of M. T. Hagan and M. Menhaj [9], which concludes that the Marquardt method is the best choice in most occasions. In next chapter, we will find out how the learning process discussed here can apply to practical applications, such as system identification and control.

CHAPTER IV

SYSTEM IDENTIFICATION AND CONTROL USING STATIC LEARNING

There are two kinds of the learning algorithms -- static and dynamic. The main theme in this chapter is the static learning process introduced in Chapter 2. The discussion of dynamic learning will be left to Chapters 5 and 6. Compared to static learning, dynamic learning uses a more complicated process to calculate the derivatives of the target function. Basically, the static learning algorithm only applies to situations in which an off-line static function approximation can be performed. One such situation is system identification (or system modeling). This situation will be discussed in the first half of this chapter.

Static learning can also be used to control dynamic systems. This process will be described in the second half of this chapter. We will present two control schemes, feedback linearization and the direct inverse controller, to demonstrate that neural network controllers can be trained off-line with static learning. A pendulum system, where the pendulum swings between two equilibrium points, will be the common computer simulation example used throughout this chapter and the rest of this document.

System Representations

For their generality and ability to use neural networks for identification, three nonlinear models, which are the system representations for a single-input/single-output (SISO) nonlinear plant, are presented. These models are generalized from their linear counterparts, which have been used in adaptive control applications for the modeling of linear systems. The three models are: (1) the nonlinear moving-average (MA) model, (2) the nonlinear autoregressive (AR) model and (3) the nonlinear autoregressive moving-average (ARMA) model.

Before the descriptions of the three models, first we define the tapped delay line (TDL) introduced in Chapter 2. The TDL is a component which produces outputs that are delayed values of the input. It is shown in Figure 4.1.

Nonlinear MA Model

In the linear moving average model the output is a moving average of the current and previous inputs, as in

$$y(k+1) = \sum_{i=0}^{n} \alpha_i u(k-i), \tag{4.1}$$

where α_i represents the impulse response of the system. The nonlinear extension to the moving average model is

$$y(k+1) = f[u(k), u(k-1), \ldots u(k-n)] \tag{4.2}$$

and the nonlinear model is shown in Figure 4.2.

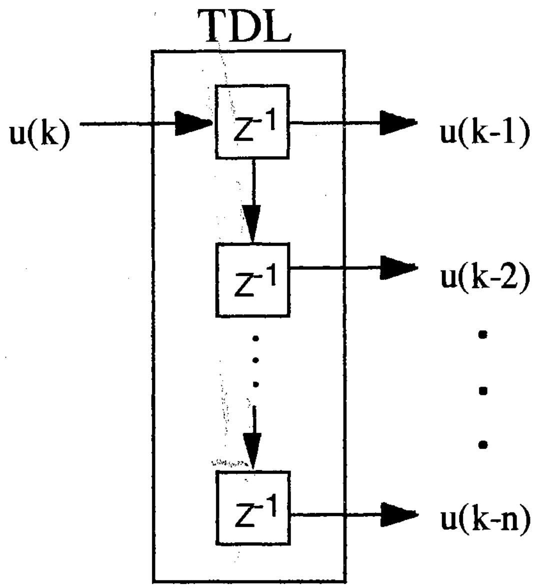


Figure 4.1 The Tapped Delay Line

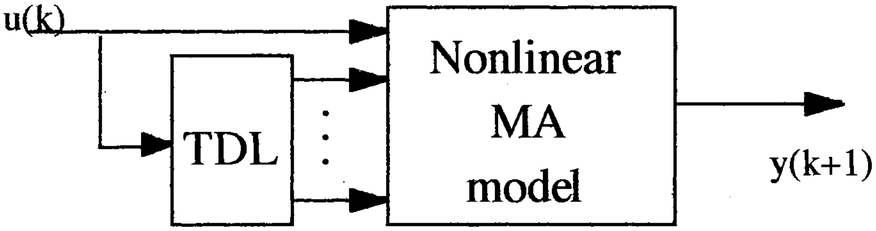


Figure 4.2 A SISO Nonlinear MA Model

Nonlinear AR Model

Another well-known model used in the representation of linear systems is the autoregressive model. The output of the linear AR model, at time stage k+1, is related linearly to its own past values

$$y(k+1) = \sum_{i=0}^{m} \alpha_i y(k-i) + u(k) \tag{4.3}$$

The nonlinear version of Equation 4.3 is

$$y(k+1) = f[y(k), y(k-1), \ldots y(k-m)] + u(k) \tag{4.4}$$

and the nonlinear model is shown in Figure 4.3.

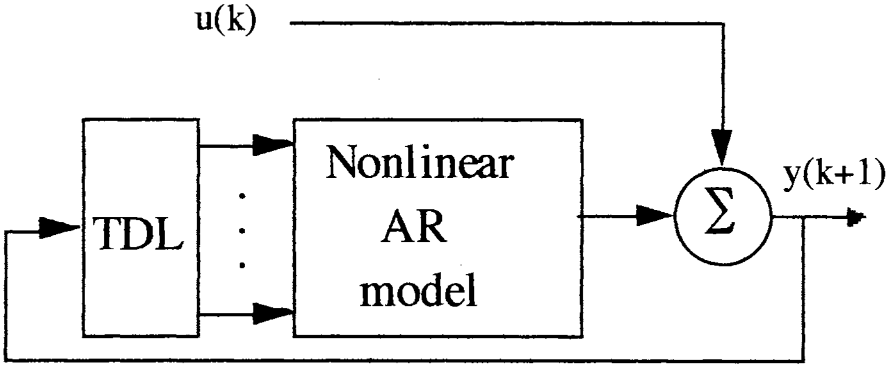


Figure 4.3 A SISO Nonlinear AR Model

Nonlinear ARMA Model

The most extensively used model in control systems is the ARMA model. This model combines the moving-average part (Equation 4.2) and the autoregressive part (Equation 4.4) from the previously discussed models and is expressed as

$$\begin{aligned} y(k+1) = h[&u(k), u(k-1), \ldots u(k-n), \\ &y(k), y(k-1), \ldots y(k-m)] \end{aligned} \qquad (4.5)$$

A nonlinear SISO ARMA model is shown in Figure 4.4.

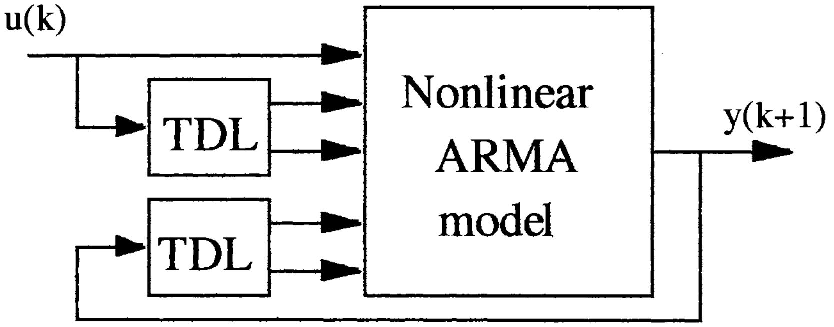


Figure 4.4 A SISO Nonlinear ARMA Model

System Identification

The three models discussed in the last section actually can be represented by just one model -- the ARMA model. Both the MA model and the AR model then become special cases of the ARMA model. Therefore, we will only need to describe the system identification for the nonlinear ARMA model in this section.

The architecture to train a neural network to model a nonlinear SISO ARMA plant is shown in Figure 4.5. The inputs to the neural network NN_p are the current plant input u(k) and the past values from both the plant input u(k) and the plant output y(k+1). The error e(k+1), which is the difference between the plant output $y_t(k+1)$ and the network output $y_{nn}(k+1)$, will be used in the static learning algorithm for adjusting the parameters of the network.

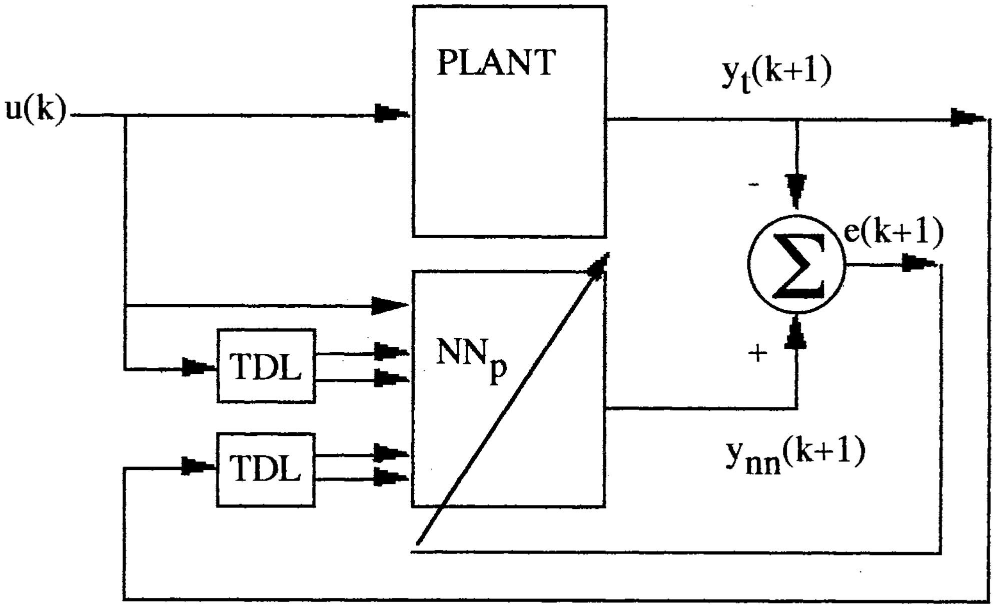


Figure 4.5 A Nonlinear SISO System Identification Using a Neural Network

The Swinging Pendulum System

A pendulum system, as shown in Figure 4.6, will be used as an example of a nonlinear SISO system for the computer simulation.

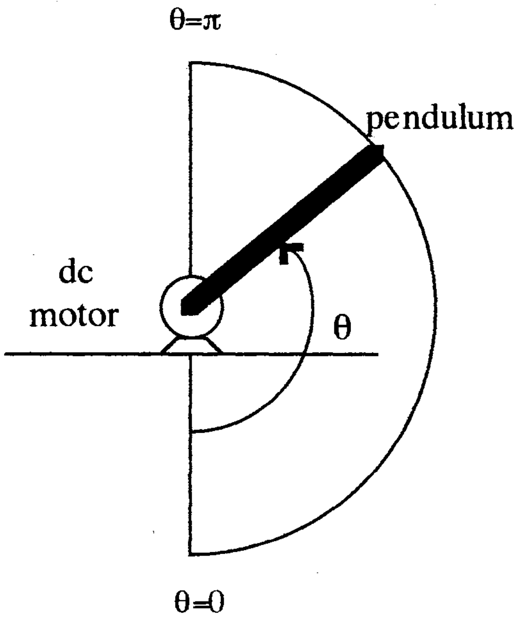


Figure 4.6 A Swinging Pendulum System

The pendulum has a full range of swing angle θ from the straight downward position (θ=0) to the straight upward position (θ=π) and is driven by a dc motor with one of its ends attached to the motor shaft. The mathematical model for the pendulum system is assumed

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -10\sin(x_1) - 2x_2 + u \end{bmatrix} \tag{4.6}$$

where $x_1=\theta$, $x_2=\dfrac{d\theta}{dt}$ and u is the current applied to the motor. The approximate discrete time system would be described by

$$\underline{x}(k+1) = \underline{x}(k) + \Delta t\begin{bmatrix} x_2(k) \\ -10\sin(x_1(k)) - 2x_2(k) + u(k) \end{bmatrix} \tag{4.7}$$

The above equation can be expressed as

$$\begin{aligned} &x_1(k+1) = f[x_1(k), u(k), x_2(k)], \\ &x_2(k) = g[x_1(k-1), u(k-1), x_2(k-1)] \end{aligned} \tag{4.8}$$

Furthermore, with the recurrent relationship shown in Equation 4.8, the pendulum system can be expressed as a nonlinear SISO ARMA model

$$x_1(k+1) = f[x_1(k), x_1(k-1), \ldots u(k), u(k-1), \ldots] \tag{4.9}$$

For the purpose of modeling the pendulum system with neural networks, we have truncated Equation 4.9 to

$$x_1(k+1) = f(x_1(k), x_1(k-1), u(k), u(k-1)) \tag{4.10}$$

Computer Simulations

To produce the training data, a sinusoidal baseline current u is applied to the motor

$$u(k) = \sin[(\Delta t)(k)], \tag{4.11}$$

where Δt (0.05 second) is the sampling time and k is the time stage index number. The pendulum begins at a randomly chosen initial position between 0 and π. If the controlled pendulum approaches the boundaries ($\theta=0$ or $\theta=\pi$), the baseline current is changed to a constant threshold current to prevent crossing the boundary. Once the pendulum is brought back to its normal swinging range by the constant threshold current, the baseline current will resume. A total of 400 input-output pairs [u(k), y(k+1)] were collected.

The alternating training data sets (ATS) method, proposed by the author, is employed in all the static learning applications in this research and will be discussed further in Chapter 9. The learning curve shown in Figure 4.7 was obtained after 10,000 learning epochs. Note that the learning curve approaches a constant value after approximately the 750-th learning epoch. This could mean that a global minimum is reached.

A series-parallel test method is adopted to evaluate the trained neural network NN_p. In this test the feedback to the network is not from the network itself but from the plant output. The series-parallel test is illustrated in Figure 4.8.

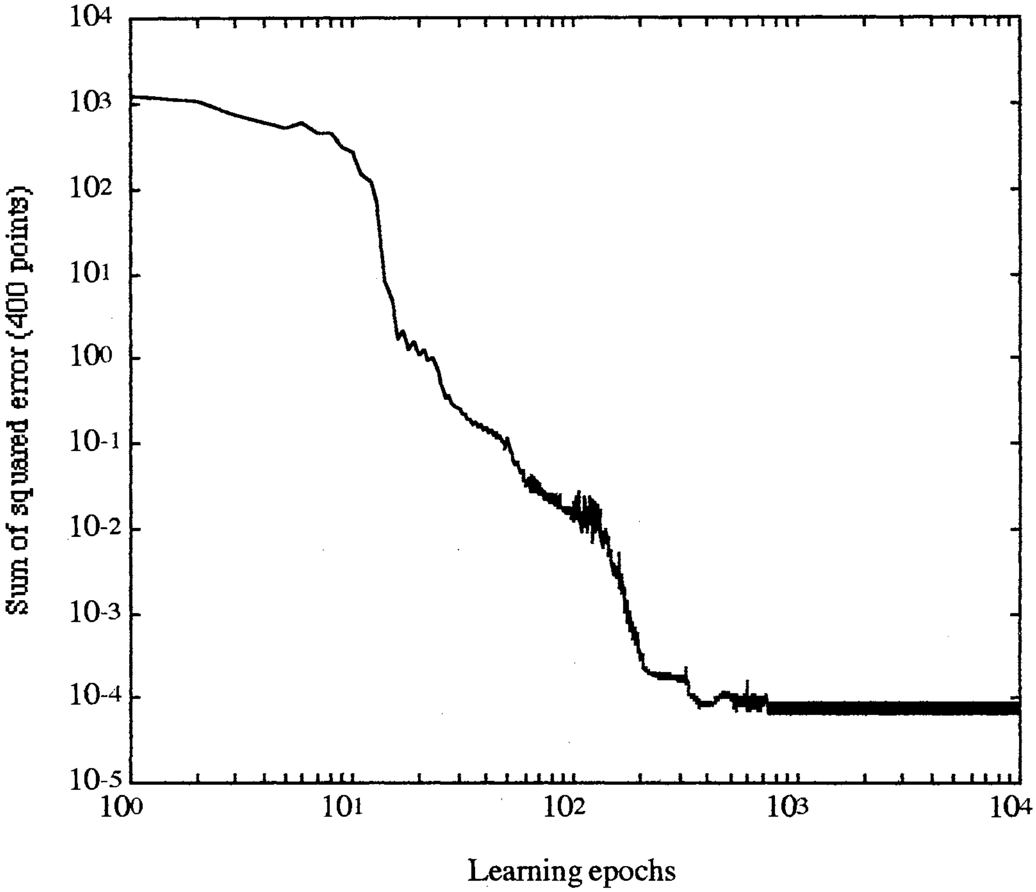


Figure 4.7 The Learning Curve of the Network NN_p

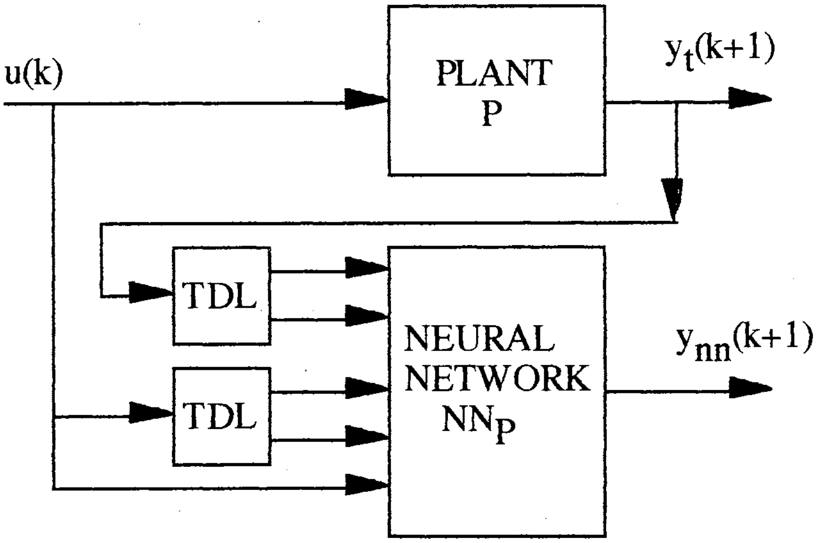


Figure 4.8 The Series-Parallel Test Method

Several tests were executed to make the evaluation and the results are shown in Figure 4.9. In each test, the initial position y(0) and the constant input current u, were different. Note that, in the graphs, a solid line would represent the plant output and a dashed line, the network output. However, due to the successful neural network modeling, only one data line appears in the each graph of Figure 4.9.

A higher precision evaluation called the parallel test, which will be described in Chapter 7, was also performed. The results obtained from the parallet test were approximately the same as the results from the series-parallel test.

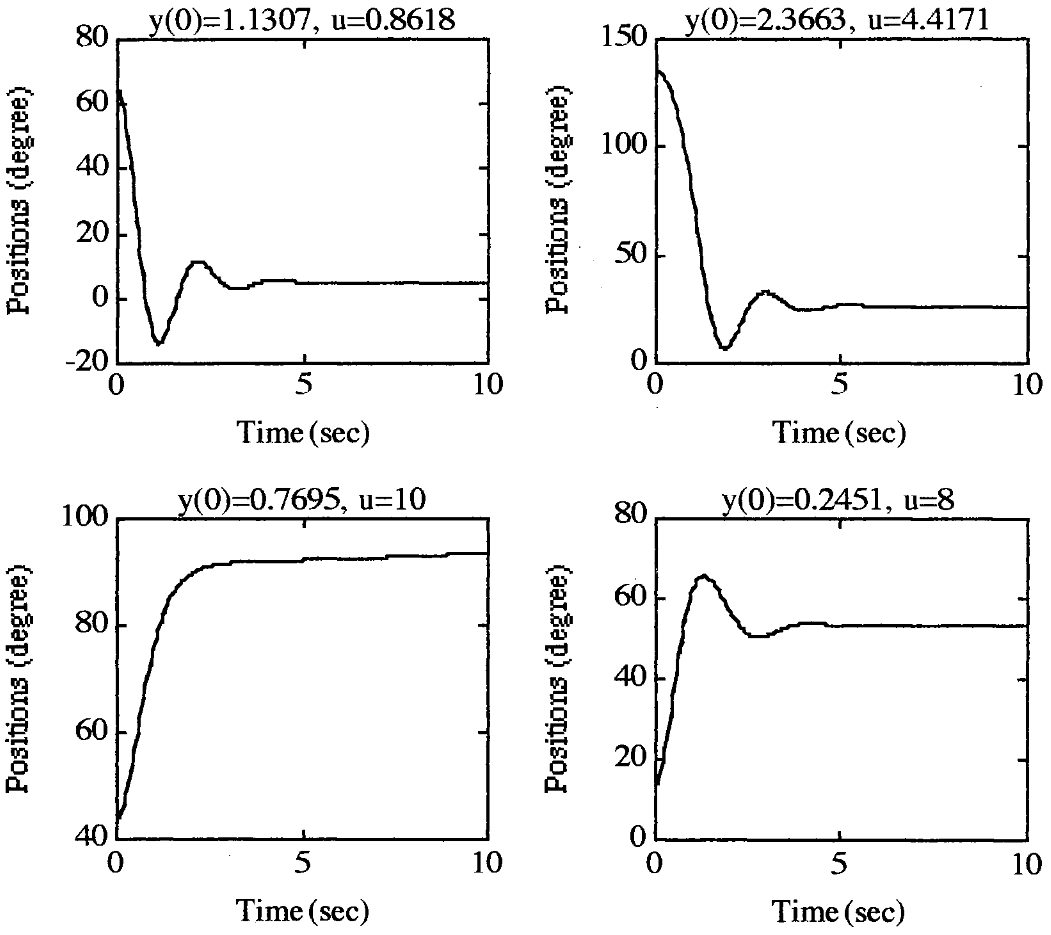


Figure 4.9 The Evaluation of the Network NN_f

Feedback Linearization [8]

Some neural network controllers can be trained using static learning; two such controllers will be discussed in this document. In this section we present the first of these two methods, feedback linearization. Consider a nonlinear SISO system which has the dynamics

$$\frac{d^n x}{dt^n} = f(\underline{x}) + b(\underline{x})u \qquad (4.12)$$

where f($\underline{x}$) is the nonlinearity, b is the constant, u is the control input, x is the output and $\underline{x}$ is the state vector which can be expressed as

$$\underline{x} = [x \ \frac{dx}{dt} \cdots \frac{d^{n-1}x}{dt^{n-1}}]^T \qquad (4.13)$$

By combining Equations 4.12 and 4.13, a state space representation is obtained

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} x_2 \\ x_3 \\ \vdots \\ x_n \\ f(\underline{x}) + b(\underline{x})u \end{bmatrix} \qquad (4.14)$$

Thus the system is in controllability canonical form. When a control input

$$u = \frac{1}{b(\underline{x})}[-\underline{k}^T \underline{x} - f(\underline{x})] \qquad (4.15)$$

is applied to this system, the nonlinearity f($\underline{x}$) is canceled and the closed loop dynamics become

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} x_2 \\ x_3 \\ \vdots \\ x_n \\ -k_1x_1 - k_2x_2 - \cdots - k_nx_n \end{bmatrix} \qquad (4.16)$$

In other words, by choosing $\underline{k}$ appropriately, the nonlinear system responds just like any desired n-th order linear system. This defines the feedback linearization.

Feedback Linearization with Neural Networks

The role that a neural network plays in feedback linearization is to replace the nonlinearity f($\underline{x}$)

$$NN_f \cong f(\underline{x}) \tag{4.17}$$

where NN_f is the neural network model of f($\underline{x}$). For the case of pendulum system, which is described by Equation 4.6, f($\underline{x}$)= -10sin(x_1)-2x_2. To find the discrete representation for this nonlinearity, Equation 4.7 is rewritten as

$$\begin{aligned}\underline{x}(k+1) &= \underline{x}(k) + \Delta t\begin{bmatrix} x_2(k) \\ -10\sin(x_1(k)) - 2x_2(k) + u(k)\end{bmatrix}. \\ &= \underline{fd}(x(k)) + \underline{gd}(u(k))\end{aligned} \tag{4.18}$$

The neural network needs to learn the second element of the function $\underline{fd}$ in order to cancel the nonlinearity. This discrete form of the nonlinearity f($\underline{x}$) would be

$$\underline{fd}_2(\underline{x}) = x_2(k) + \Delta t[-10\sin(x_1(k)) - x_2(k)] \tag{4.19}$$

Suppose that we would like the pendulum closed loop system to respond with the dynamics given by

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -9x_1 - 6x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 9r \end{bmatrix}, \tag{4.20}$$

Then, from Equation 4.15, the continuous feedback linearization controller would be

$$u_{FL} = 9r - [9 \;\; 6]\underline{x} - f(\underline{x}) \tag{4.21}$$

and the control input in a discrete form is

$$u_{FL}(k) = \frac{x_2(k) + \Delta t(9r(k) - [9 \;\; 6]\underline{x}(k)) - NN_f}{\Delta t} \tag{4.22}$$

Computer Simulations for Nonlinearity Neural Network

The first step in feedback linearization control is to train the nonlinearity neural network NN_f. By empirical experience, a 2-8-1_33 multilayer feedforward network is chosen for this purpose. Assume that x(1) and x(2) fall into the intervals $[0,\pi]$ and $[-2\pi,2\pi]$ respectively.

The process for obtaining the training data set is the same as that described in the section on system identification. However, in this case Equation 4.19 is used for the process. In the simulation, the sampling time is 0.05 second, and the total number of data points in the training data set is 400. The ATS method is used in the training. The resulting learning curve is shown in Figure 4.10. Like the learning curve in system identification, the learning curve in this simulation also approaches a constant value which suggests that a global minimum has been reached.

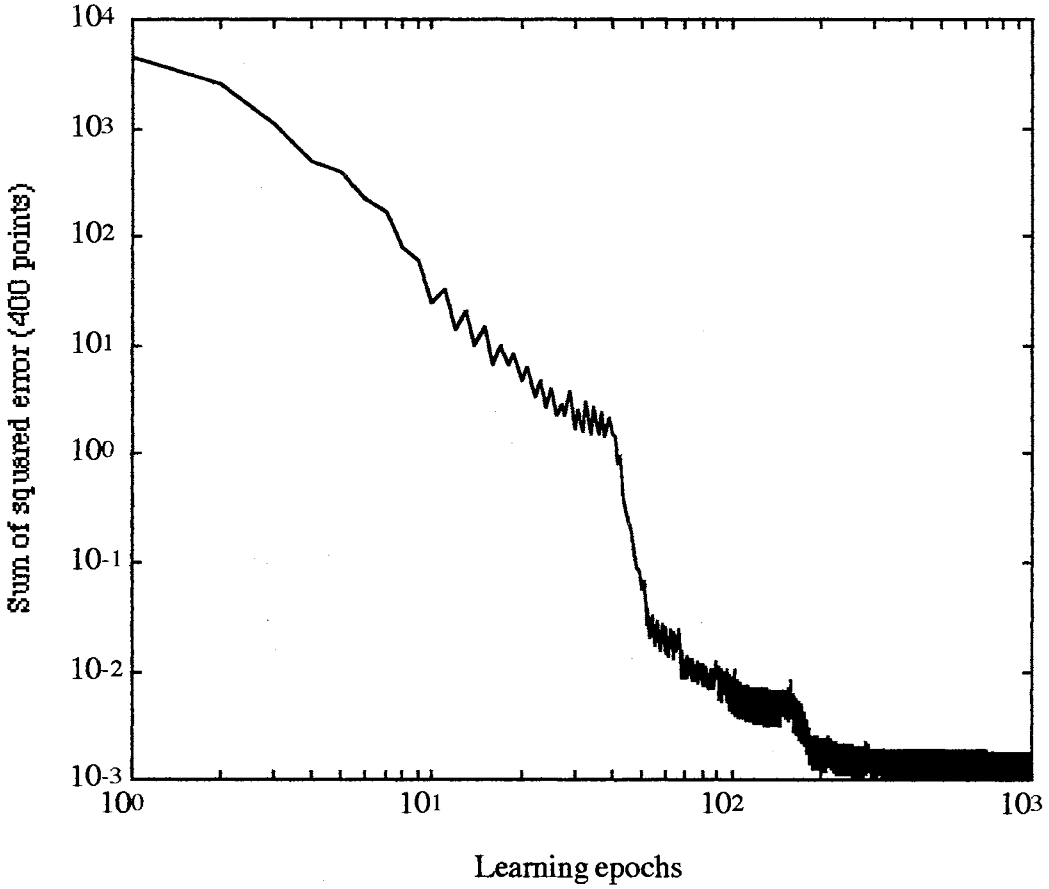


Figure 4.10 The learning curve of the trained Network NN_f

Two tests were performed to evaluate the trained network NN_f. The first test was executed under the following conditions -- x(1) was varied while x(2) was set to zero. Then, in the second test the conditions were reversed and x(2) was varied while x(1) was set to zero. The results of the tests are shown in Figure 4.11. Although, in the graphics, the network output is represented by the dashed line while the true linearity is represented by the solid line, one can not really distinguish between them. Once again, this is a result of very accurate approximation by the network NN_f.

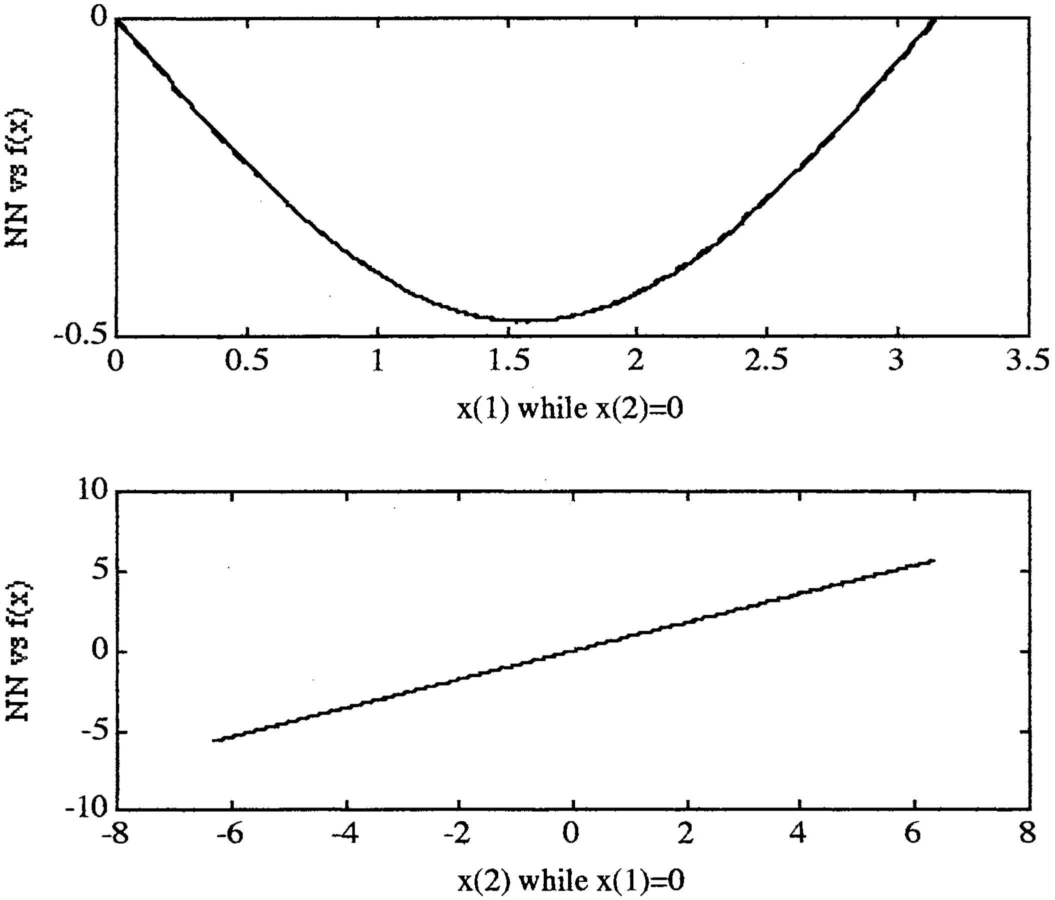


Figure 4.11 The Evaluation of the Network NN_f

Computer Simulations for Feedback Linearization with Neural Network

Next, we will evaluate the performance of the feedback linearization controller where the trained network NN_f is embedded in it. For comparison, the performance of a linear controller is also given. The linear controller was designed for the case where the pendulum model is linearized about the state $\underline{x}$=[$\pi/2$, 0]

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -10 - 2x_2 + u \end{bmatrix} \qquad (4.23)$$

The linear controller which would cause the linearized pendulum system to respond as the reference model (Equation 4.20), is

$$u_{LIN} = 10 + 9r - [9\ 4]\underline{x} \qquad (4.24)$$

Given a common reference position of $\pi/2$ and two initial positions, 0 and π, both the feedback linearization controller and the linear controller were tested against the reference model. The results are shown in Figure 4.12. Since the linear model is linearized about the point $\underline{x}$=[$\pi/2$, 0], the corresponding linear controller can be expected to perform well around the position $\pi/2$. However, it is observed from Figure 4.12, that the linear controller performs well as a regulator, but poorly in the tracking situation. In comparison, the feedback linearization controller shows a strong capability for tracking but fails to maintain the same position as the reference model in the final steady state.

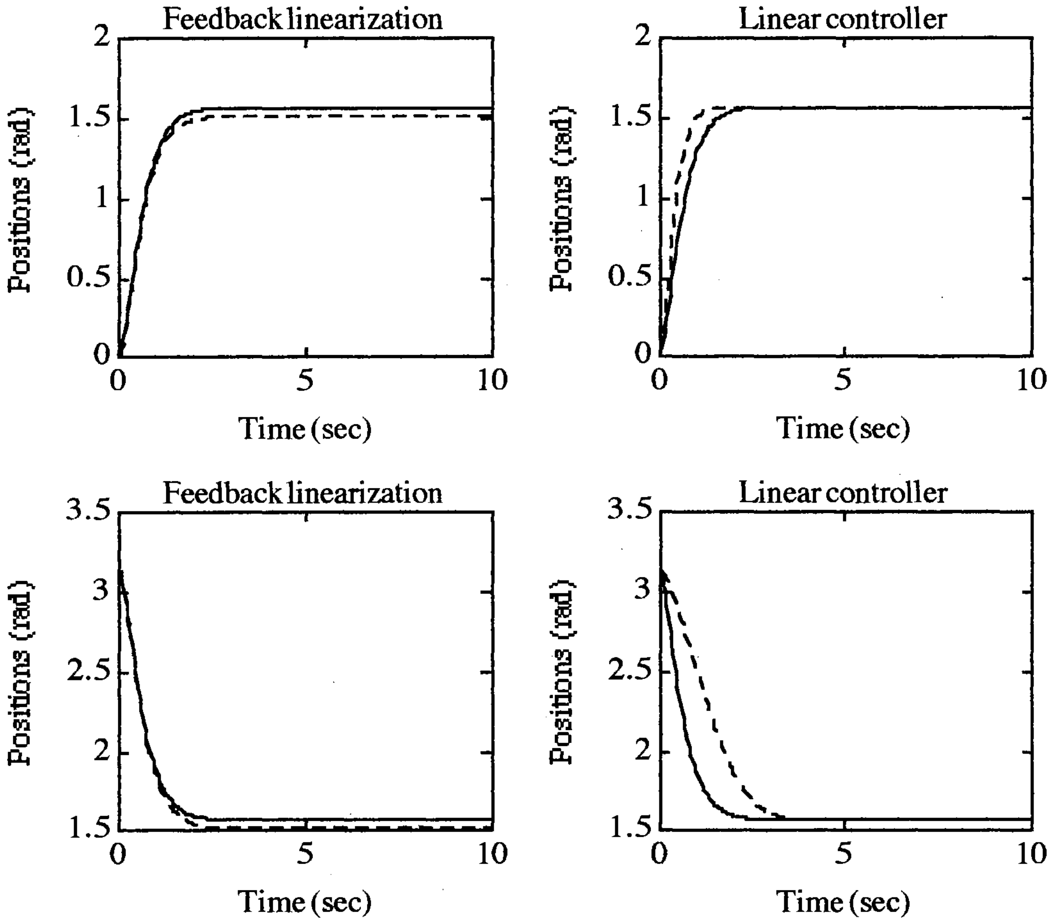


Figure 4.12 Feedback Linearization Vs Linear Controller (#1)

Two more similar tests were performed and the results are shown in Figure 4.13. In these two tests, instead of using some pre-selected values, the initial positions and the reference positions are randomly chosen between 0 and π. It seems, from the results shown in Figure 4.13, the feedback linearization controller has better performance than the linear controller in both tracking and regulating.

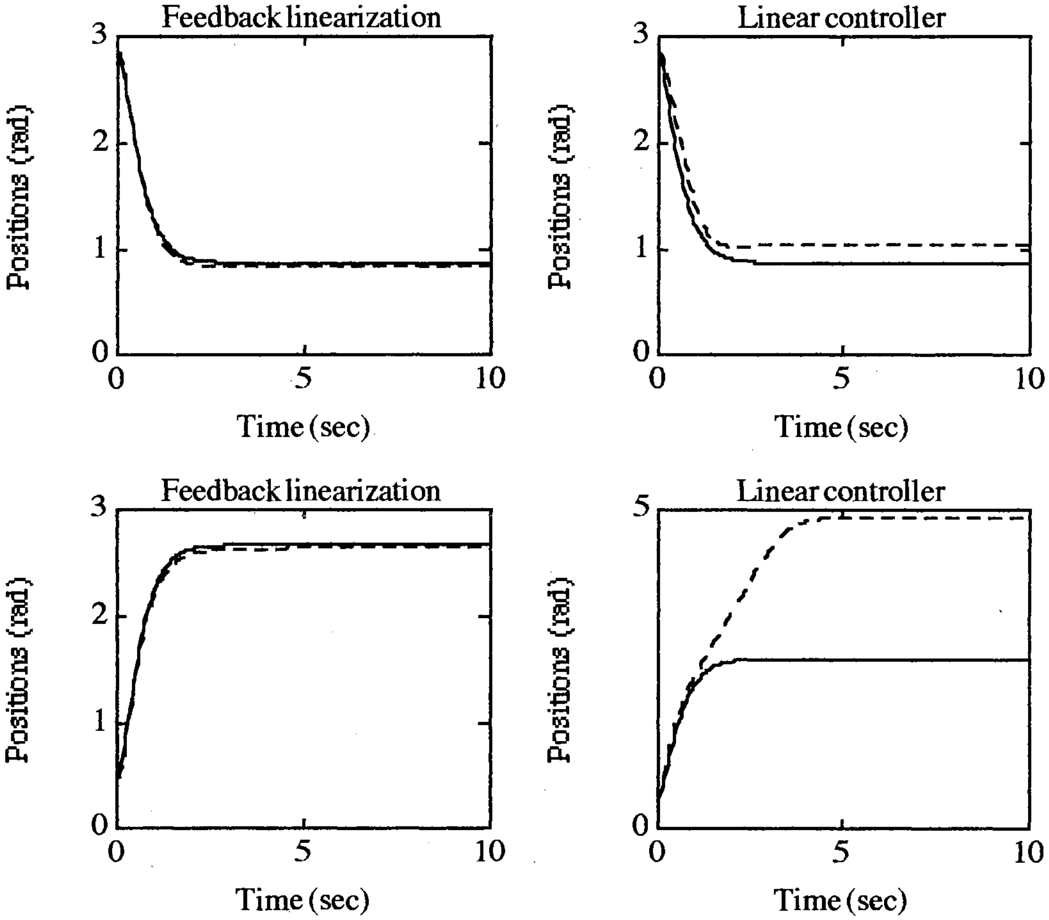


Figure 4.13 Feedback Linearization Vs Linear Controller (#2)

Direct Inverse Control

Another type of neural network control is called direct inverse control. The key step in direct inverse control is to model the inverse dynamics of a plant, in which the order of the input and the output of the plant are reversed, as shown in Figure 4.14.

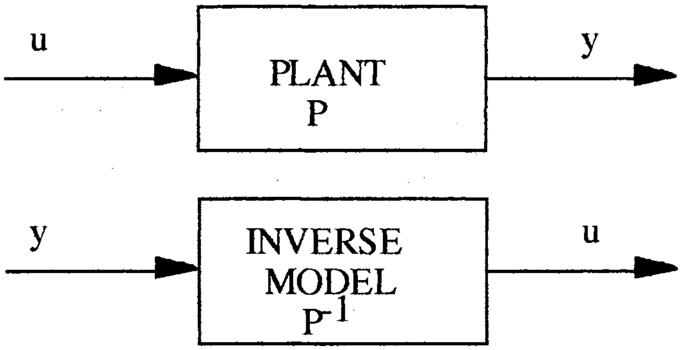


Figure 4.14 The Inverse Modeling

Once the inverse model has been identified, it can be used as a controller to manipulate the plant under conditions in which the desired output of the plant is the input of the controller. This is called the direct inverse control method and is shown in Figure 4.15.

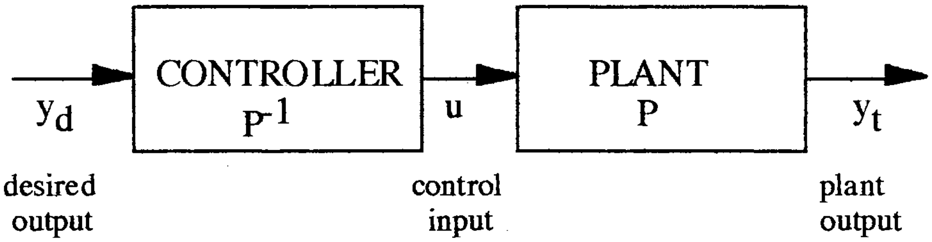


Figure 4.15 Direct Inverse Control

AMA Method

For consistency, the pendulum system, which was used in the system identification and the feedback linearization sections, again will be used in the computer simulation for the direct inverse control. Equation 4.10, which is the ARMA model of the pendulum system, is rearranged as follows

$$u(k) = g[u(k-1), y(k-1), y(k), y(k+1)], \qquad (4.25)$$

This is now the ARMA model for the inverse pendulum system. We have attempted to model Equation 4.25 using neural networks but have had no success so far. However, an approximate MA model (AMA) method, proposed by the author, has been successfully employed to model the inverse pendulum system using neural networks. It is known that an ARMA model can be represented by an infinite MA model. In this case, Equation 4.25 would then change to

$$u(k) = g[y(0), y(1), \ldots y(k-1), y(k), y(k+1)] \qquad (4.26)$$

For modeling with neural networks, Equation 4.26 is truncated as

$$u(k) = h[y(k-n), y(k-n+1), \ldots y(k-1), y(k), y(k+1)], (4.27)$$

where n is chosen by trial-and-error, in the absence of a more systematic method (a future research subject).

A 10-10-1_121 feedforward multilayer network was picked for this experiment. The parameter n in Equation 4.26 is chosen to be 8. The architecture to train the neural network model NN_p^{-1} for the inverse pendulum plant is shown in Figure 4.16.

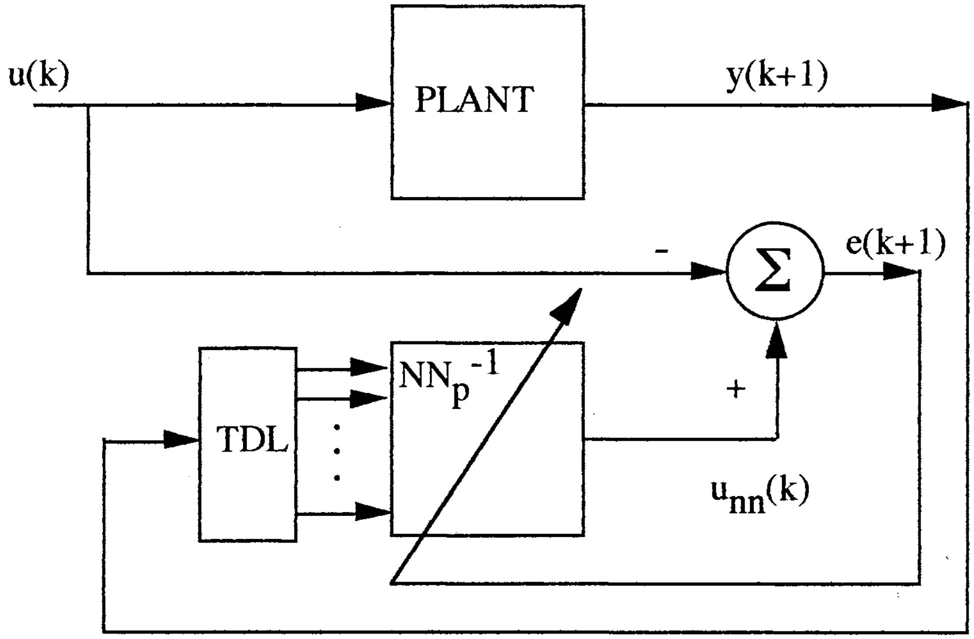


Figure 4.16 Modeling Inverse Plant with a Neural Network

Computer Simulations

The training data set was obtained by stacking the data points that resulted from several runs, up a total number of 1000. In each run, the pendulum started from a randomly chosen initial position between $\pi/4$ and $3\pi/4$. Then, a constant current, whose value was randomly chosen between -20 and 20, was applied to the motor. A run stopped when the pendulum hit the boundaries ($\theta=0$ and $\theta=\pi$), or a maximum of 50 data points were collected. The ATS method was used with the static training algorithm. The learning curve of the network NN_p^{-1} was obtained after 10,000 learning epochs and is shown in Figure 4.17.

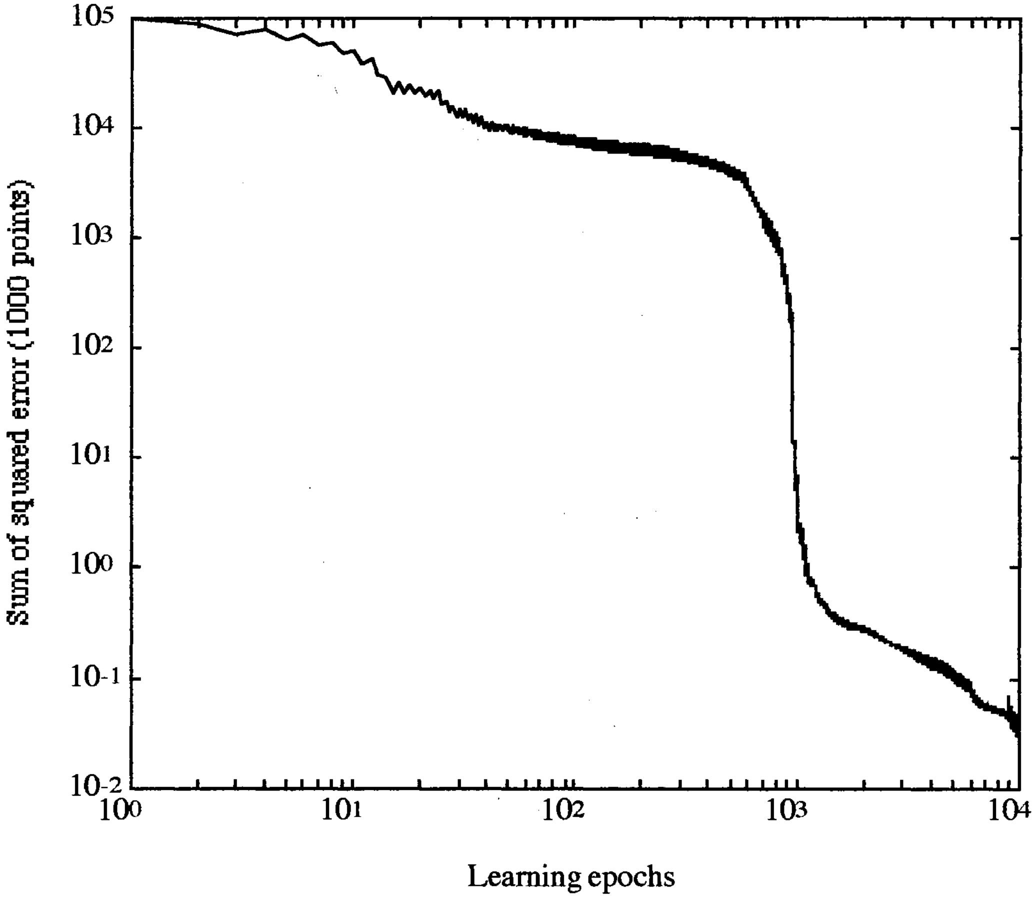


Figure 4.17 The Learning Curve of the Network NN_p^{-1}

The reference model described in Equation 4.20 supplies the desired position as the input for the direct inverse controller. The trained network NN_p^{-1} is evaluated through several tests. A randomly chosen initial position y(0) and a steady state reference position r were given for each test. The results were collected and are shown in Figure 4.18. The solid line represents the reference model output and the dashed line is the network plant output.

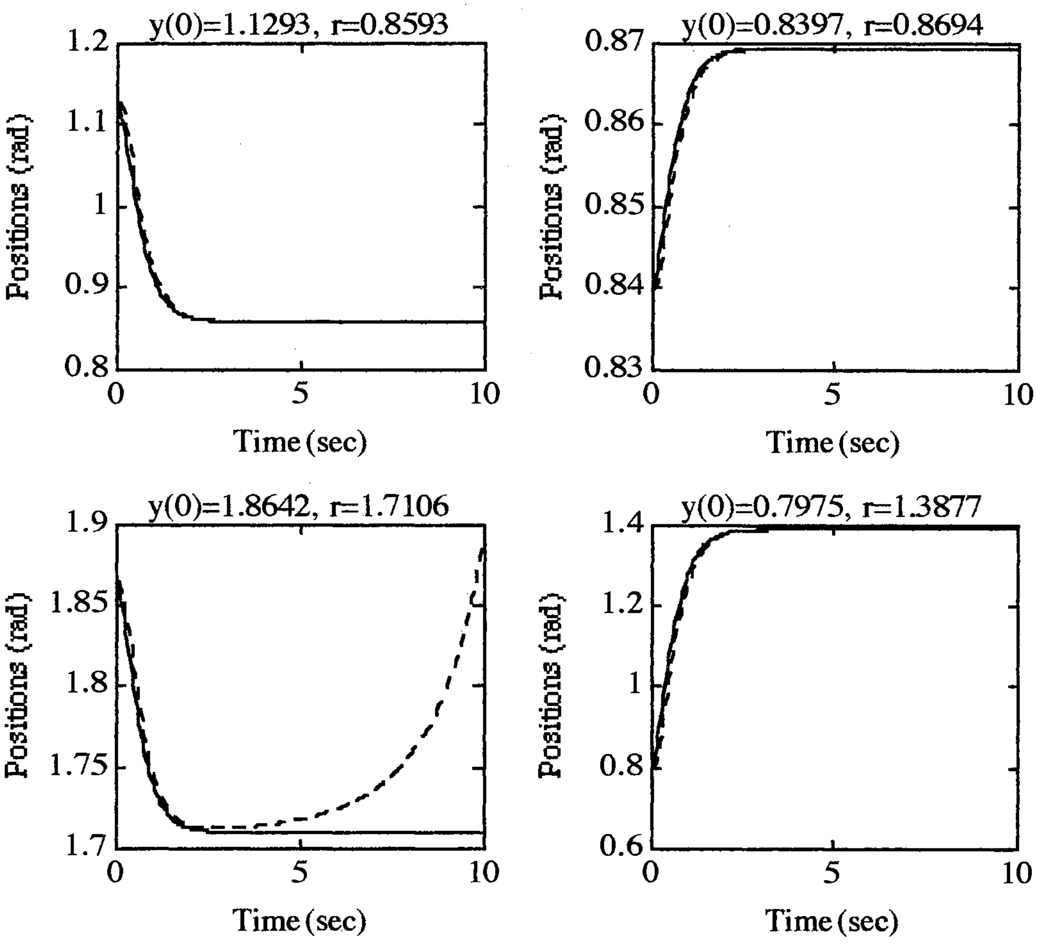


Figure 4.18 The Evaluation Tests of the Network NN_p^{-1}

The conclusion from the evaluation tests is that the trained network NN_p^{-1} performed well in most cases, but some fine tuning of the network parameters is still needed. The key to performance may be in whether or not the training data set completely represents all of the input and output spaces. This question will be discussed further in the dissertation as one of the subjects in the sensitivity analysis of the training parameters. Nevertheless, during the tests the ability of the direct inverse controller NN_p^{-1} to follow a reference model, which was never included in the learning process, was suprising.

Summary

The main point we have attempted to establish in this chapter is that the static learning process, besides its most common application -- system identification -- can also be used to train neural network controllers. This is demonstrated with feedback linearization and direct inverse control. The direct inverse control was suprisingly successful, nevertheless, the fact that the static learning process can only be performed under the assumption that no recurrent connections exist in the network limits its application. To deal with this disadvantage, a dynamic learning algorithm, which can be implemented with the existence of the recurrent connections, will be introduced and discussed in the next chapter.

CHAPTER V

FORWARD PERTURBATION

For real time adaptive identification and control applications that use neural networks, it is inevitable that some applications will need a supervised learning algorithm other than the static learning algorithm described previously in Chapter 4. Since the time dimension is added in this new learning process, it is called supervised dynamic learning. Just as static learning was used to train multilayer feedforward networks, dynamic learning is used to train recurrent networks. These recurrent networks will be described in the first section of this chapter. The concept of the recurrent connection between the neurons of the network is also defined in the same section. Following that, a description of the dynamic learning process is given, in which two important derivative calculation equations are derived. The derivation of these equations leads to two known dynamic learning algorithms -- forward perturbation (FP) and backpropagation-through-time (BTT).

Recurrent Networks

There are many different types of neural network architectures. The multilayer feedforward network, which was described in

Chapter 2 and is shown again in Figure 5.1, is one of the most commonly used architectures. In Chapter 4, we discussed and demonstrated this network and trained it with the static learning algorithm.

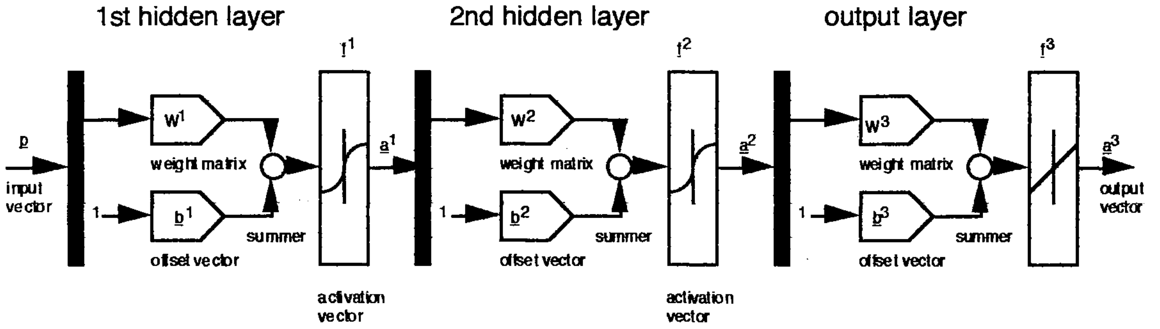


Figure 5.1 A MIMO Two Hidden Layer Network

Another commonly used network architecture is called the recurrent network. A recurrent network exists if at least one of the neurons of the network has feedback. Feedback means that, in the network, there are recurrent connections, either direct or indirect, between one neuron and the neurons from the same or other networks. This recurrent connection is illustrated in Figure 5.2. In the figure, the block which takes the output from the neuron can be constructed by any type of network structure. It can be made up of a neuron, a layer of neurons, a part of the network, a whole network, two connected networks, or simply a void.

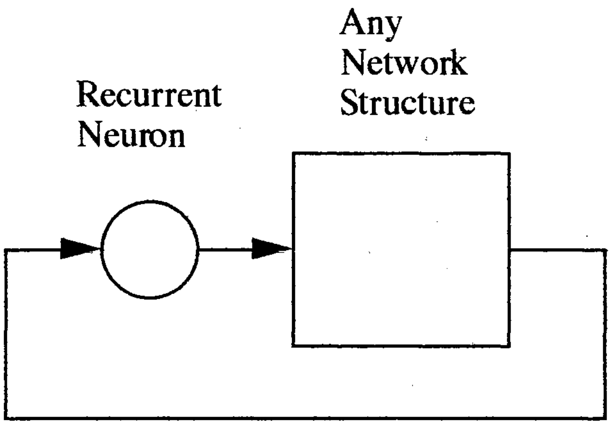


Figure 5.2 A Recurrent Connection

The multilayer feedforward network with a TDL component, which was described in Chapter 2, is shown again in Figure 5.3. Since the TDL connects the network's output to its input, the network has a recurrent connection. Thus, this network is a recurrent network. We will use this type of recurrent network throughout this document in our discussion of supervised dynamic learning.

The fact that the output of a recurrent network depends not only on its present input, but also on its past output, gives it an edge in performance over the feedforward network. However, this additional time dimension is also the reason that the learning process in a recurrent network is much more complex than the learning process in a feedforward network. We will explain this complexity in the following section.

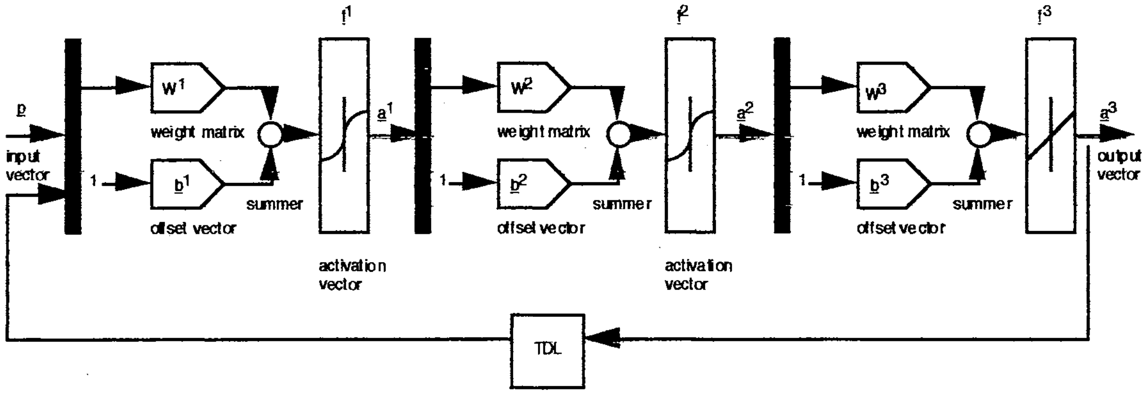


Figure 5.3 Recurrent Multilayer Network

Dynamic Learning

In static learning, the training data is collected before the learning epoch begins and is never changed during the learning process. In Chapter 4 we used this method to train the pendulum identifier and the feedback linearization and direct inverse controllers. The disadvantage of the static learning process is obvious. If the parameters of a system are changed, or if more noise is added to the application environment, the static identifiers and controllers will no longer perform the job well. In this situation, there is a need to collect a new set of training data from the application and to retrain the identifiers and controllers.

To counteract this disadvantage, we present an on-line adaptive learning process called supervised dynamic learning. In supervised dynamic learning, the recurrent network is placed on-line with its application. This is illustrated in Figure 5.4, where an example of a dynamic forward identification training configuration is given.

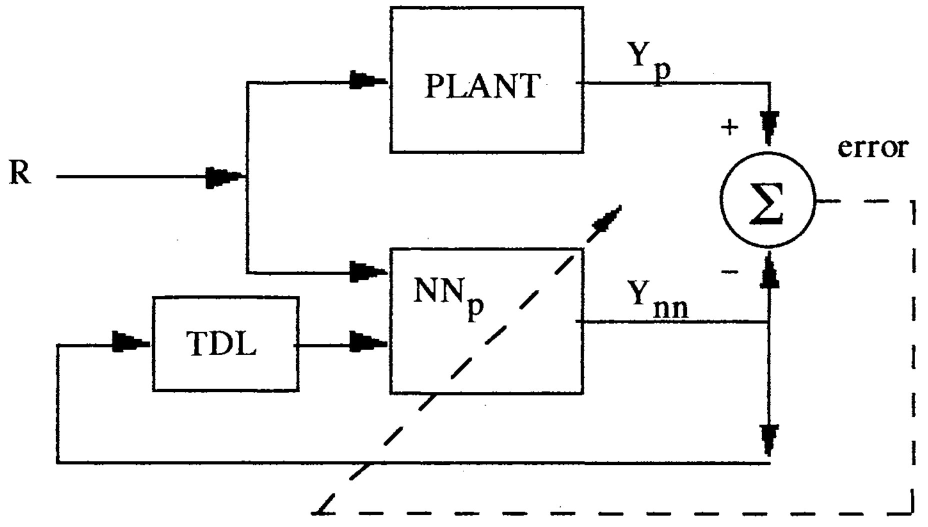


Figure 5.4 Example of A Recurrent Network Application

In comparison, the plant is not needed in the static learning process once the training data set has been collected. Also, the static learning and dynamic learning processes obtain the training data set differently. In general, the training data for static learning is never changed during the training and the training data for dynamic learning will change as learning epoch proceeds. We will discuss this in further detail in Chapter 7.

Calculation of the True Derivative

A key procedure in the supervised learning process is to compute the derivative of the objective function with respect to the network parameters. Suppose we want to find the complete derivative of the objective function, $F(\underline{a}_k(\underline{x}),\underline{x})$, with respect to the variable $\underline{x}$ through time, where k is the discrete time stage index. This complete derivative is called the true derivative in this document. Then, applying the chain rule to the objective function with respect to the variables $\underline{a}_k$ and $\underline{x}$, the true derivative of the objective function can be obtained either from

$$\frac{\partial F}{\partial \underline{x}} = \sum_k \frac{\partial \underline{a}_k}{\partial \underline{x}}^T \frac{\partial^e F}{\partial \underline{a}_k} \tag{5.1}$$

or from

$$\frac{\partial F}{\partial \underline{x}} = \sum_k \frac{\partial^e \underline{a}_k}{\partial \underline{x}}^T \frac{\partial F}{\partial \underline{a}_k} \tag{5.2}$$

where the superscript e stands for the explicit derivative. The explicit derivative terms in both Equations 5.1 and 5.2 can be obtained using the basic backpropagation method we already described and demonstrated in the static learning discussion in Chapters 3 and 4. The implicit derivative term in Equation 5.1, $\frac{\partial \underline{a}_k}{\partial \underline{x}}$, is the perturbation-through-time (PTT) at the k-th time stage. The implicit derivative term in Equation 5.2, $\frac{\partial F}{\partial \underline{a}_k}$, is the sensitivity-through-time (STT) at the k-th time stage. To obtain the PTT and STT, we apply the chain rule again through time. This results in:

$$\frac{\partial \underline{a}_k}{\partial \underline{x}} = \frac{\partial^e \underline{a}_k}{\partial \underline{x}} + \frac{\partial^e \underline{a}_k}{\partial \underline{a}_{k-1}} \frac{\partial \underline{a}_{k-1}}{\partial \underline{x}} \tag{5.3}$$

and

$$\frac{\partial F}{\partial \underline{a}_k} = \frac{\partial^e F}{\partial \underline{a}_k} + \frac{\partial^e \underline{a}_{k+1}}{\partial \underline{a}_k}^T \frac{\partial F}{\partial \underline{a}_{k+1}} \tag{5.4}$$

Equations 5.1 and 5.3 make up the FP algorithm and Equations 5.2 and 5.4 make up the BTT algorithm. Note that in Equation 5.3 $\frac{\partial \underline{a}_k}{\partial \underline{x}}$ is computed from $\frac{\partial \underline{a}_{k-1}}{\partial \underline{x}}$, which explains why this is called the forward perturbation method. We see that in Equation 5.4 $\frac{\partial F}{\partial \underline{a}_k}$ is computed from $\frac{\partial F}{\partial \underline{a}_{k+1}}$, which is why this method is called backpropagation through time. The FP algorithm will be further discussed in the following section and a discussion of the BTT algorithm can be found in the next chapter.

Forward Perturbation

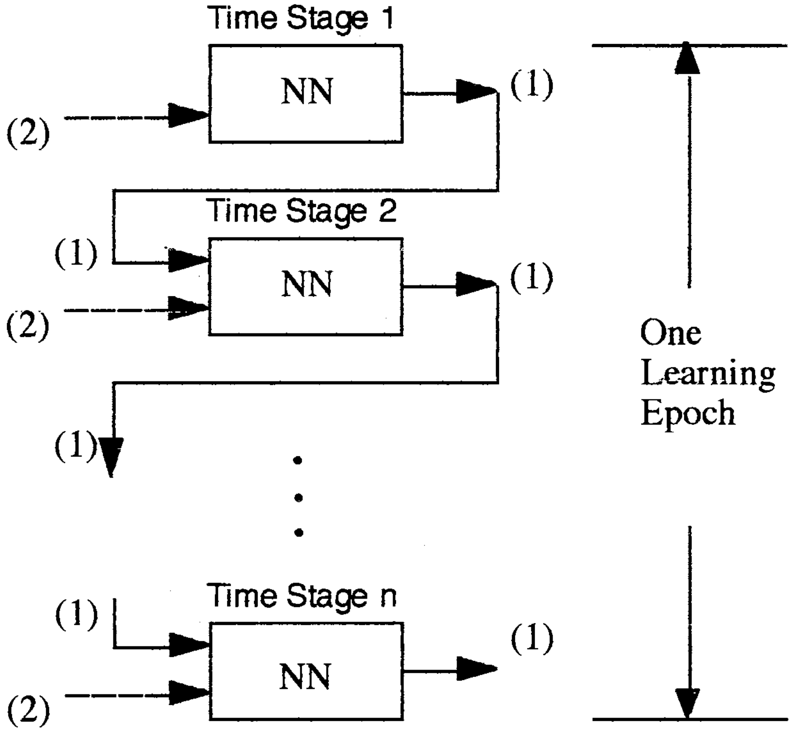
We start this section with a discussion of previous research and an introduction to recurrent and non-recurrent variables in dynamic learning. Then, two basic dynamic learning models are discussed and their corresponding equations are derived. The equations calculate the true derivatives of the objective function with respect to the network parameters through time. This discussion is followed by an illustrative example, which demonstrates the derivative calculations using one of the basic FP equations. In the example, the actual derivatives are also calculated by hand to verify the results from the FP calculation. Then, a couple of more complex examples of FP calculations are discussed. Finally, to illustrate the use of Marquardt optimization in dynamic learning, the computations and the requirements for both the Jacobian matrix of the error function and the derivative vector of the objective function are explained.

Previous Work on FP

As we mentioned in the first section of this chapter, although the added time dimension in the recurrent network strengthens its performance, it also complicates the calculation of the derivative. This is the reason that forward perturbation, which in Narendra's work [10] is also called dynamic backpropagation, has not yet been widely understood and used. The term "forward perturbation" is excerpted from a paper by Werbos [6], which describes its basic concept. A full description of the FP algorithm can be found in the Narendra paper [11]. Several neural network representations, which are the building blocks for recurrent network applications in dynamic learning, are also discussed in another Narendra paper [12].

Recurrent and Non-recurrent Variables

In a recurrent network, the recurrent variable is defined as a function of the network parameters and the previous values of some of the network variables. For example, the outputs of the recurrent network are recurrent variables. The other variables in a recurrent network, which have not met the definition of a recurrent variable, are called non-recurrent variables. The external inputs of a recurrent network are examples of non-recurrent variables. From the viewpoint of one learning epoch, recurrent variables and non-recurrent variables are illustrated in Figure 5.5 .



(1) Recurrent variables
(2) Non-recurrent variables

Figure 5.5 Recurrent and Non-recurrent Variables

The introduction of recurrent and non-recurrent variables are essential to our description of the true derivative and its calculation using the FP algorithm described in the following section.

Basic Dynamic Learning Models and True Derivative Calculation

Two basic dynamic learning models are proposed in this section. We will use the FP algorithm to derive the true derivative calculation equations for each of the models. The basic dynamic learning Model I is a MISO (multi-input single-output) recurrent network as shown in Figure 5.6. The output of the model y(k) is a recurrent variable. If the elements of the proceeding input vector $\underline{x}(k)$ are all non-recurrent variables, then the derivative of the output with respect to the network parameter vector $\underline{w}$, $\frac{\partial y(k)}{\partial \underline{w}}$, can be obtained with the basic backpropagation method described and demonstrated in Chapters 3 and 4.

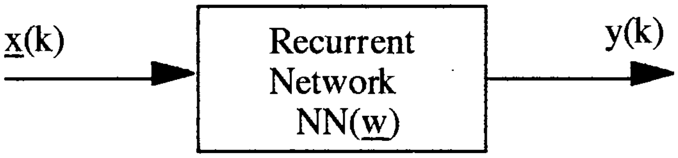


Figure 5.6 Dynamic Learning Model I

If the input vector $\underline{x}(k)$ in Figure 5.6 can be decomposed into $\underline{x}_{nr}$ and $\underline{x}_r$, which stand for the non-recurrent and recurrent variables respectively, then, from Equation 5.3, the PTT is calculated as

$$\frac{\partial y(k)}{\partial \underline{w}} = \frac{\partial^e y(k)}{\partial \underline{w}} + \frac{\partial^e y(k)}{\partial \underline{x}_r(k)}^T \frac{\partial \underline{x}_r(k)}{\partial \underline{w}} \tag{5.5}$$

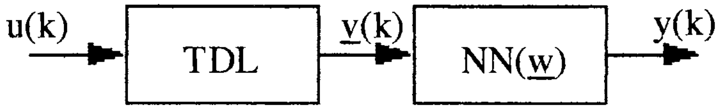
where the explicit derivative $\frac{\partial^e y(k)}{\partial \underline{w}}$ deals with both the recurrent and non-recurrent variables. As for the implicit derivative, $\frac{\partial \underline{x}_r(k)}{\partial \underline{w}}$ in Equation 5.5, it represents the PTT from the previous time stages and can be calculated using Equation 5.3.

The basic dynamic learning Model II is shown in Figure 5.7. In this model a multi-output TDL component, which has the following output

$$\underline{v}(k) = [u(k-1)\ u(k-2)\ \ldots u(k-m)]^T, \tag{5.6}$$

is connected to a network NN($\underline{w}$). We assume that the input of the TDL is a recurrent variable. Thus, by combining Equations 5.5 and 5.6, the PTT in this case is

$$\begin{aligned}\frac{\partial y(k)}{\partial \underline{w}} &= \frac{\partial y^e(k)}{\partial \underline{w}} + \frac{\partial^e y(k)}{\partial \underline{v}(k)}^T \frac{\partial \underline{v}(k)}{\partial \underline{w}} \\ &= \frac{\partial y^e(k)}{\partial \underline{w}} + \sum_{j=1}^{m} \frac{\partial^e y(k)}{\partial u(k-j)} \frac{\partial u(k-j)}{\partial \underline{w}}\end{aligned} \tag{5.7}$$



$$\underline{v}(k) = [u(k\text{-}1)\ u(k\text{-}2)\ \dots\ u(k\text{-}m)]^T$$

Figure 5.7 Dynamic Learning Model II

An Illustrative Example

The simple recurrent network, shown in Figure 5.8, will be used to demonstrate the calculation of the true derivative using the FP dynamic learning algorithm.

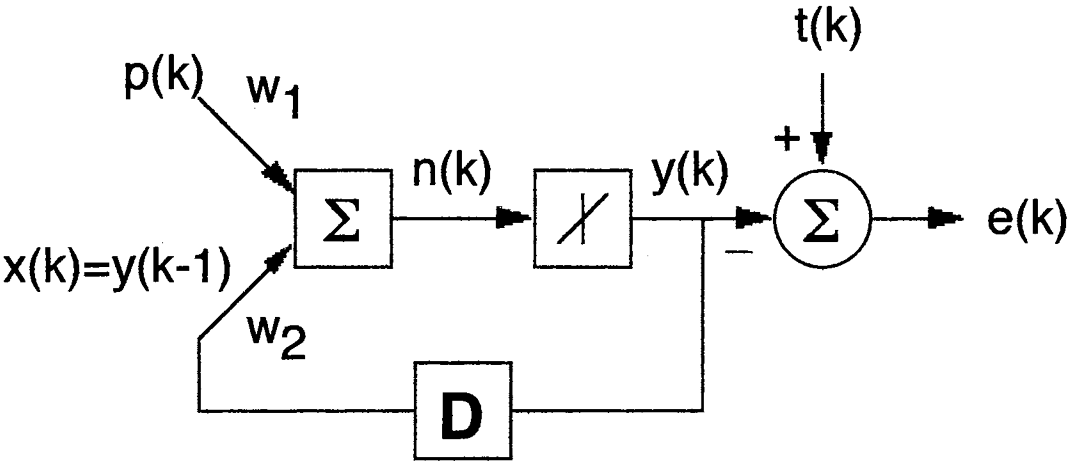


Figure 5.8 A Simple Recurrent Network

In Figure 5.8, the recurrent network is connected to the external input p(k) and the recurrent input x(k) through the connecting weights w_1 and w_2 ($\underline{w}=[w_1\ w_2]^T$) respectively. The offset is set to zero and the activation function of the network is linear. Three forward time stages (from k=1 to k=3) for the recurrent network are performed and the results are

$$\begin{aligned}
y(1) &= w_1p(1) + w_2x(1) \\
&= w_1p(1) \\
y(2) &= w_1p(2) + w_2x(2) \\
&= w_1p(2) + w_2y(1) \\
&= w_1p(2) + w_2w_1p(1) \\
y(3) &= w_1p(3) + w_2x(3) \\
&= w_1p(3) + w_2y(2) \\
&= w_1p(3) + w_2w_1p(2) + w_2{}^2w_1p(1)
\end{aligned} \tag{5.8}$$

The objective function is defined as

$$F = \frac{1}{2}\sum_{k=1}^{3}(t(k) - y(k))^2 \tag{5.9}$$

To calculate the true derivatives of the objective function with respect to the network parameter vector $\underline{w}$ we use Equation 5.1:

$$\frac{\partial F}{\partial \underline{w}} = \sum_{k=1}^{3}\frac{\partial y(k)}{\partial \underline{w}}\frac{\partial^e F}{\partial y(k)}. \tag{5.10}$$

We start with the calculation of the PTT $\frac{\partial y(k)}{\partial \underline{w}}$ using Equation 5.7. In this case, the number of the TDL outputs, m, is one. Thus, the PTT at time stage k for this example is

$$\begin{aligned}\frac{\partial y(k)}{\partial \underline{w}} &= \frac{\partial^e y(k)}{\partial \underline{w}} + \frac{\partial x(k)}{\partial \underline{w}}\frac{\partial^e y(k)}{\partial x(k)} \\ &= \frac{\partial^e y(k)}{\partial \underline{w}} + \frac{\partial y(k-1)}{\partial \underline{w}}\frac{\partial^e y(k)}{\partial y(k-1)}\end{aligned} \tag{5.11}$$

Using Equation 5.11, the three forward time stages are also calculated in the following set of equations;

k=1,

$$\begin{aligned}\frac{\partial y(1)}{\partial \underline{w}} &= \frac{\partial^e y(1)}{\partial \underline{w}} + \frac{\partial y(0)}{\partial \underline{w}}\frac{\partial^e y(1)}{\partial y(0)} \\ &= \frac{\partial^e y(1)}{\partial \underline{w}} = \left[p(1) \quad 0\right]^T;\end{aligned} \tag{5.11}$$

k=2,

$$\begin{aligned}\frac{\partial y(2)}{\partial \underline{w}} &= \frac{\partial^e y(2)}{\partial \underline{w}} + \frac{\partial y(1)}{\partial \underline{w}}\frac{\partial^e y(2)}{\partial y(1)} \\ &= \frac{\partial^e y(2)}{\partial \underline{w}} + \frac{\partial y(1)}{\partial \underline{w}}\frac{\partial^e y(2)}{\partial y(1)} \\ &= \left[p(2) \quad x(2)\right]^T + w_2\left[p(1) \quad 0\right]^T \\ &= \left[p(2)+w_2p(1) \quad w_1p(1)\right]^T;\end{aligned} \tag{5.12}$$

k=3,

$$\begin{aligned}\frac{\partial y(3)}{\partial \underline{w}} &= \frac{\partial^e y(3)}{\partial \underline{w}} + \frac{\partial y(2)}{\partial \underline{w}}\frac{\partial^e y(3)}{\partial y(2)} \\ &= \frac{\partial^e y(3)}{\partial \underline{w}} + \frac{\partial y(2)}{\partial \underline{w}}\frac{\partial^e y(3)}{\partial y(2)} \\ &= \left[p(3) \quad x(3)\right]^T + w_2\left[p(2)+w_2p(1) \quad w_1p(1)\right]^T \\ &= \begin{bmatrix} p(3)+w_2p(2)+w_2{}^2p(1) \\ w_1p(2)+2w_2w_1p(1) \end{bmatrix};\end{aligned} \tag{5.13}$$

Once the PTTs are obtained we can substitute them into Equation 5.10 to obtain the true derivative of the objective function w.r.t. the parameter vector $\underline{w}$

$$\begin{aligned}\frac{\partial F}{\partial \underline{w}} &= \frac{\partial y(1)}{\partial \underline{w}}\frac{\partial^e F}{\partial y(1)} + \frac{\partial y(2)}{\partial \underline{w}}\frac{\partial^e F}{\partial y(2)} + \frac{\partial y(3)}{\partial \underline{w}}\frac{\partial^e F}{\partial y(3)} \\ &= -\begin{bmatrix} p(1)(t(1)-y(1)) \\ 0 \end{bmatrix} - \\ &\quad \begin{bmatrix} (p(2)+w_2p(1))(t(2)-y(2)) \\ w_1p(1)(t(2)-y(2)) \end{bmatrix} - \\ &\quad \begin{bmatrix} (p(3)+w_2p(2)+w_2{}^2p(1))(t(3)-y(3)) \\ (w_1p(2)+2w_2w_1p(1))(t(3)-y(3)) \end{bmatrix}\end{aligned} \tag{5.14}$$

For comparison, we can simply calculate the actual derivatives by hand using Equation 5.8. The results are

$$\begin{aligned}\frac{\partial F}{\partial w_1} &= -[p(1)(t(1)-y(1)) + \\ &\quad (p(2)+w_2p(1))(t(2)-y(2)) + \\ &\quad (p(3)+w_2p(2)+w_2{}^2p(1))(t(3)-y(3))], \\ \frac{\partial F}{\partial w_2} &= -[w_1p(1)(t(2)-y(2)) + \\ &\quad (w_1p(2)+2w_2w_1p(1))(t(3)-y(3))].\end{aligned} \tag{5.15}$$

which agrees with Equation 5.14. We can observe from Equation 5.14 that the FP algorithm not only produces the true derivative over one learning epoch but also generates the true derivative for each time stage in a learning epoch. This unique characteristic of the FP algorithm is very important in Chapter 6 where the FP and the BTT algorithms are compared.

Complex Examples in Dynamic Learning

In order to demonstrate the applicability of the two basic dynamic learning models to any recurrent network configuration, two more complex examples are given in the following. In addition to the training network, a non-training neural network (where the parameters are held constant) is also involved in both cases.

In Figure 5.9, the network to be trained is NN($\underline{w}$) and the non-training network is NN'. Since the configuration is a combination of the two previously described basic dynamic learning models, both Equation 5.5 and 5.6 are used to calculate the network perturbations. For simplicity, we will only calculate the key term in the FP algorithm -- PTT -- from here on. We start with calculating the PTT of the non-training network at time stage k,

$$\begin{aligned}\frac{\partial y(k)}{\partial \underline{w}} &= \frac{\partial y^e(k)}{\partial \underline{w}} + \frac{\partial^e y(k)}{\partial v(k)}^T \frac{\partial v(k)}{\partial \underline{w}} \\ &= 0 + \frac{\partial^e y(k)}{\partial v(k)}^T \frac{\partial v(k)}{\partial \underline{w}}\end{aligned} \tag{5.16}$$

where $\frac{\partial y^e(k)}{\partial \underline{w}}$ is zero because y(k) is not explicitly a function of the training network parameter vector $\underline{w}$. Then, the PTT of the training network at time stage k is

$$\begin{aligned}\frac{\partial v(k)}{\partial \underline{w}} &= \frac{\partial^e v(k)}{\partial \underline{w}} + \frac{\partial u(k)}{\partial \underline{w}} \frac{\partial^e v(k)}{\partial u(k)} \\ &= \frac{\partial^e v(k)}{\partial \underline{w}} + \frac{\partial y(k-1)}{\partial \underline{w}} \frac{\partial^e v(k)}{\partial y(k-1)}\end{aligned} \tag{5.17}$$

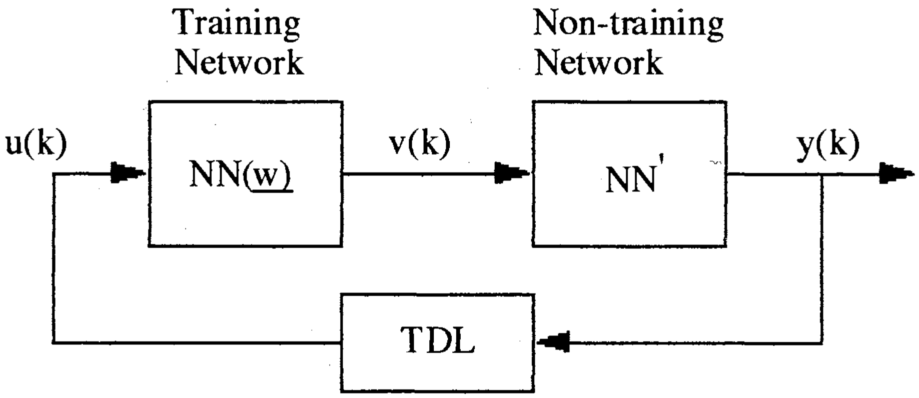


Figure 5.9 First Complex Example for Dynamic Learning

By comparing Equations 5.16 and 5.17, one can see that there is a recursive relation between these equations as the time stage moves forward in one learning epoch.

The configuration in the second example is similar to the first one. A TDL component is added between the two networks as shown in Figure 5.10. For simplicity, the TDL has only one delay output x(k). This example is still a combination of the two basic dynamic learning models. Thus, the calculation of the PTT of the non-training network at time stage k is

$$\begin{aligned}\frac{\partial y(k)}{\partial \underline{w}} &= \frac{\partial y^e(k)}{\partial \underline{w}} + \frac{\partial^e y(k)}{\partial x(k)}\frac{\partial x(k)}{\partial \underline{w}} \\ &= 0 + \frac{\partial^e y(k)}{\partial v(k-1)}\frac{\partial v(k-1)}{\partial \underline{w}}\end{aligned} \quad . \tag{5.18}$$

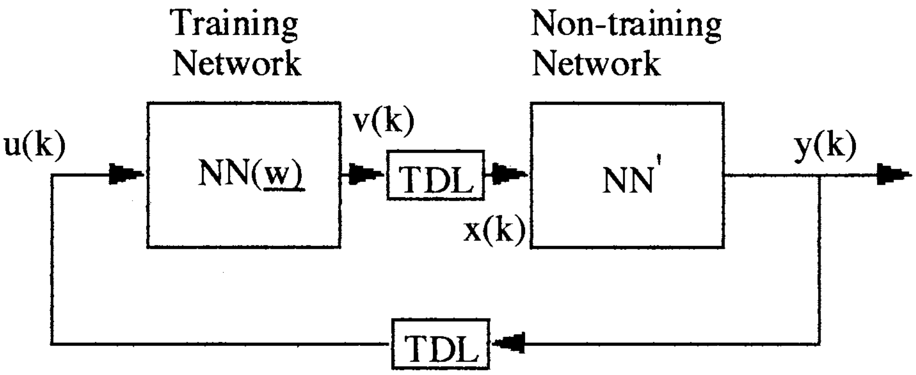


Figure 5.10 Second Complex Example for Dynamic Learning

To find out the PTT of the training network $\frac{\partial v(k-1)}{\partial \underline{w}}$ at time stage k-1 in the Equation 5.18, we start with computing the PTT of the training network at time stage k

$$\begin{aligned}\frac{\partial v(k)}{\partial \underline{w}} &= \frac{\partial^e v(k)}{\partial \underline{w}} + \frac{\partial u(k)}{\partial \underline{w}} \frac{\partial^e v(k)}{\partial u(k)} \\ &= \frac{\partial^e v(k)}{\partial \underline{w}} + \frac{\partial y(k-1)}{\partial \underline{w}} \cdot \frac{\partial^e v(k)}{\partial y(k-1)},\end{aligned} \tag{5.19}$$

Marquardt Optimization Using FP

In Chapter 3 we stated that, throughout this document, the Marquardt method will be used in the optimization stage of each supervised learning process. This claim not only applies to the static learning process, but also to the dynamic learning process. The optimization equation for the Marquardt method, which is restated in

Equation 5.20, can only be executed after a Jacobian matrix $J(\underline{w})$ is obtained

$$\Delta\underline{w} = [J(\underline{w})^T J(\underline{w}) + \mu I]^{-1} [J(\underline{w})^T \underline{f}(\underline{w})] \qquad (5.20)$$

The following Jacobian matrix is the derivative of the error function $\underline{f}$ with respect to the network parameter vector $\underline{w}$

$$J(\underline{w}) = \begin{bmatrix} \frac{\partial f_1(\underline{w})}{\partial w_1} & \frac{\partial f_1(\underline{w})}{\partial w_2} & \cdots & \frac{\partial f_1(\underline{w})}{\partial w_n} \\ \frac{\partial f_2(\underline{w})}{\partial w_1} & \frac{\partial f_2(\underline{w})}{\partial w_2} & \cdots & \frac{\partial f_2(\underline{w})}{\partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_N(\underline{w})}{\partial w_1} & \frac{\partial f_N(\underline{w})}{\partial w_2} & \cdots & \frac{\partial f_N(\underline{w})}{\partial w_n} \end{bmatrix} \qquad (5.21)$$

The error function, for a single-output network in one learning epoch, is defined as

$$\underline{f} = \underline{t} - \underline{y} \qquad (5.22)$$

where $\underline{t}$ and $\underline{y}$ are the desired output and the actual output respectively. Therefore, the derivative of $\underline{f}$ with respect to the parameter vector $\underline{w}$ can be expressed as

$$\frac{\partial \underline{f}}{\partial \underline{w}} = \frac{\partial(\underline{t} - \underline{y})}{\partial \underline{w}} = -\frac{\partial \underline{y}}{\partial \underline{w}} \qquad (5.23)$$

It is clear from this equation that, with only a sign difference, $\frac{\partial \underline{f}}{\partial \underline{w}}$ is the PTT discussed previously . Note that the true derivative of the objective function is never needed in the Marquardt optimization method. Thus, it is a perfect match to use the FP algorithm and the Marquardt optimization in dynamic learning.

Summary

As the chapter title suggests, the main subject that has been discussed in this chapter is the FP algorithm for supervised dynamic learning. Although the basic concept and a full treatment of the algorithm can be found in the works of Werbos and Narendra, a somewhat simpler explanation of the FP algorithm is proposed here. The explanation starts with the definitions of recurrent and non-recurrent variables. Then, two basic dynamic learning models were proposed, which actually were derived from the definition of the recurrent variable. The corresponding equations for calculating the PTTs of both the basic models are described. Through an illustrative example and two complex examples, the applicability of the two basic dynamic models to complex recurrent network configurations is demonstrated. This demonstration continues in Chapter 7 where the FP algorithm is implemented in the model reference adaptive control (MRAC) method, which serves as an example of control using dynamic learning.

CHAPTER VI

BACKPROPAGATION-THROUGH-TIME

Another dynamic learning algorithm is backpropagation-through-time (BTT). As explained in Chapter 5, both the BTT and FP algorithms originate from the same principle. It is the manner in which the chain rule is implemented which distinguished the two algotithms. We will present the BTT algorithm in the same sequence as we laid out for our discussion of the FP algorithm. It begins with a reiteration of the derivation of the BTT derivative equation and STT calculation equations, which were previously presented in Chapter 5. This is followed by application of BTT to the same illustrative example and complex examples we used in the FP algorithm discussion. The implementation of the Marquardt optimization using BTT, which is not as natural in this applications as it was in FP, is then discussed. Finally, based on computer simulations, comparisons between the FP and BTT algorithms are given to show the highlights of the two dynamic learning algorithms.

Derivative Calculation

As we commented at the end of the illustrative example in the FP algorithm discussions, the true derivative of the objective function can be obtained not only over one learning epoch, but also at each time stage of the learning epoch. This advantage comes from the fact that the PTT at each time stage of a learning epoch, which is accumulated from both past and present time stages, reflects the complete derivative of the network error at that point in time.

In the BTT process, we have a different situation. First, the true derivative is obtained only over each learning epoch and not at each time stage. Second, it is the sensitivity, instead of the perturbation, which carries the information that is needed in the true derivative calculation process. Finally, the sensitivities are backpropagated not just through time (from future time stages to the present time stage), but also through each layer of the network for the present time stage. We will explain all of the above differences in the following section.

The BTT Derivative Calculation Equations

Recall from Chapter 5 that Equations 5.2 and 5.4 define the BTT algorithm. These equations are rewritten here,

$$\frac{\partial F}{\partial \underline{w}} = \sum_{k} \frac{\partial^{e} \underline{a}_{k}}{\partial \underline{w}}^{T} \frac{\partial F}{\partial \underline{a}_{k}} \tag{6.1}$$

and

$$\frac{\partial F}{\partial \underline{a}_{k}} = \frac{\partial^{e} F}{\partial \underline{a}_{k}} + \frac{\partial^{e} \underline{a}_{k+1}}{\partial \underline{a}_{k}}^{T} \frac{\partial F}{\partial \underline{a}_{k+1}} \tag{6.2}$$

where the terms $\frac{\partial F}{\partial \underline{a}_k}$ and $\frac{\partial F}{\partial \underline{a}_{k+1}}$ are the STTs at time stages k and k+1 respectively. This means that the STT is an accumulation of the sensitivities from both the future and the present time stages.

Calculation of STT

The two basic dynamic learning models described in Chapter 5, are shown here again in Figures 6.1 and 6.2. They are used here to demonstrate how the STT is calculated. After the STT is obtained, the true derivative is then computed using Equation 6.1.

Using Equation 6.2, the STT equation for use in Model I (MISO) of the dynamic learning process can be easily obtained as,

$$\frac{\partial F}{\partial y(k)} = \frac{\partial^e F}{\partial y(k)} + \frac{\partial^e y(k+1)}{\partial y(k)} \frac{\partial F}{\partial y(k+1)} \qquad (6.3)$$

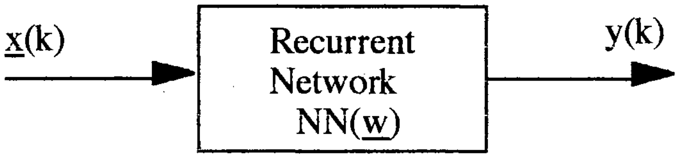
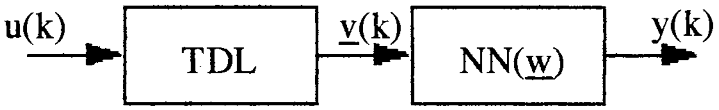


Figure 6.1 Dynamic Learning Model I

The STT equation for use in Model II of the dynamic learning process is obtained as

$$\begin{aligned}\frac{\partial F}{\partial y(k)} &= \frac{\partial^e F}{\partial y(k)} + \frac{\partial^e \underline{v}(k)}{\partial y(k)} \frac{\partial F}{\partial \underline{v}(k)} \\ &= \frac{\partial^e F}{\partial y(k)} + \sum_{j=1}^{m} \frac{\partial^e u(k+m)}{\partial y(k)} \frac{\partial F}{\partial u(k+m)}\end{aligned} \tag{6.4}$$

where m is an index from the last time stage to the current time stage in the learning epoch. Note that both of the above equations only apply to the network output layer. After the output layer STT $\frac{\partial F}{\partial \underline{a}_k}$ is obtained, the STTs for each hidden layer of the network, at time stage k, are computed using the basic backpropagation method.



$\underline{v}(k) = [u(k\text{-}1)\ u(k\text{-}2)\ ...\ u(k\text{-}m)]^T$

Figure 6.2 Dynamic Learning Model II

An Illustrative Example

The simple recurrent network used in Chapter 5, and shown again here in Figure 6.3, will be used to demonstrate the calculation of the true derivative using the BTT dynamic learning algorithm.

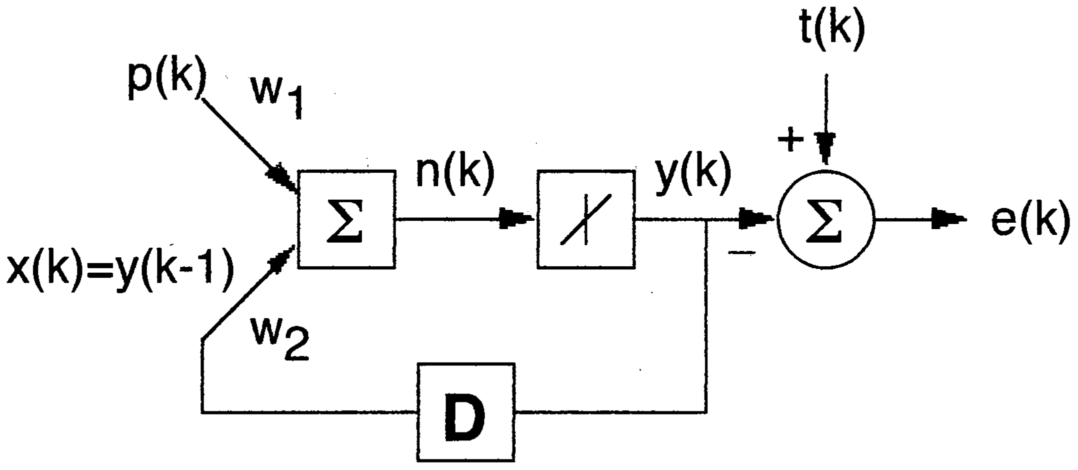


Figure 6.3 A Simple Recurrent Network

The three forward time stage operations and the objective function are rewritten here as

$$\begin{aligned} y(1) &= w_1p(1) + w_2x(1) \\ &= w_1p(1) \\ y(2) &= w_1p(2) + w_2x(2) \\ &= w_1p(2) + w_2y(1) \\ &= w_1p(2) + w_2w_1p(1) \\ y(3) &= w_1p(3) + w_2x(3) \\ &= w_1p(3) + w_2y(2) \\ &= w_1p(3) + w_2w_1p(2) + w_2^2w_1p(1) \end{aligned} \tag{6.5}$$

and

$$F = \frac{1}{2}\sum_{k=1}^{3}(t(k) - y(k))^2 \tag{6.6}$$

To calculate the true derivatives of the objective function with respect to the network parameter vector $\underline{w}$, using the BTT algorithm,

$$\frac{\partial F}{\partial \underline{w}} = \sum_{k=1}^{3} \frac{\partial y^{e}(k)}{\partial \underline{w}} \frac{\partial F}{\partial y(k)}, \tag{6.7}$$

we must first start with the calculation of the STT $\frac{\partial y(k)}{\partial \underline{w}}$ using Equation 6.3. Since F is not a function of y(4), the STTs from k=3 backward to k=1 are

k=3,

$$\begin{aligned} \frac{\partial F}{\partial y(3)} &= \frac{\partial^{e} F}{\partial y(3)} + \frac{\partial^{e} y(4)}{\partial y(3)} \frac{\partial F}{\partial y(4)} \\ &= -1 \cdot (t(3) - y(3)) + 0 \\ &= -(t(3) - y(3)); \end{aligned} \tag{6.8}$$

k=2,

$$\begin{aligned} \frac{\partial F}{\partial y(2)} &= \frac{\partial^{e} F}{\partial y(2)} + \frac{\partial^{e} y(3)}{\partial y(2)} \frac{\partial F}{\partial y(3)} \\ &= -(1)(t(2) - y(2)) - (w_2)(t(3) - y(3)) \\ &= -(t(2) - y(2)) - w_2(t(3) - y(3)); \end{aligned} \tag{6.9}$$

k=1,

$$\begin{aligned} \frac{\partial F}{\partial y(1)} &= \frac{\partial^{e} f(1)}{\partial y(1)} + \frac{\partial^{e} y(2)}{\partial y(1)} \frac{\partial F}{\partial y(2)} \\ &= -(1)(t(1) - y(1)) - \\ &\quad (w_2)(t(2) - y(2) + w_2(t(3) - y(3))) \\ &= -(t(1) - y(1)) - \\ &\quad [w_2(t(2) - y(2)) + w_2^{2}(t(3) - y(3))]; \end{aligned} \tag{6.10}$$

Then, from Equation 6.1, the true derivative of the objective function w.r.t. the weight vector over the three time stages is calculated using

$$\begin{aligned}\frac{\partial F}{\partial \underline{w}} &= \sum_{k=1}^{3} \frac{\partial^{e} y(k)}{\partial \underline{w}} \frac{\partial F}{\partial y(k)} \\ &= \frac{\partial^{e} y(1)}{\partial \underline{w}} \frac{\partial F}{\partial y(1)} + \frac{\partial^{e} y(2)}{\partial \underline{w}} \frac{\partial F}{\partial y(2)} + \frac{\partial^{e} y(3)}{\partial \underline{w}} \frac{\partial F}{\partial y(3)} \\ &= \begin{bmatrix} p(1) \\ 0 \end{bmatrix} [-(t(1)-y(1)) - w_2(t(2)-y(2)) \\ &\qquad - w_2^{2}(t(3)-y(3))] + \\ &\quad \begin{bmatrix} p(2) \\ y(1) \end{bmatrix} [-(t(2)-y(2)) - w_2(t(3)-y(3)] + \qquad (6.11) \\ &\quad \begin{bmatrix} p(3) \\ y(2) \end{bmatrix} [-(t(3)-y(3))]. \end{aligned}$$

By comparing the results from Equation 6.11 with the results from Equation 5.15, the true derivative, it is found that they agree, as expected.

An important point we would like to stress here, is that, after comparing Equation 6.11 with Equation 5.14, we note that the STTs at each time stage are different with the corresponding PTTs at each time stage. As we observed in Equation 6.11, this difference results from the fact that the STT contains the error information from the future time stages. The calculation of the PTT at each time stage in the FP algorithm is consistent with the fact that the error that occurred in the network output at each time stage results from both the past and the present inputs. In contrast, the calculation of the STT in the BTT algorithm only looks at the impact in the present and future from present point of view. This explains why only the PTT and not the STT can be converted directly into the true derivative at each time stage. As for the STT, a more complex calculation has to be performed before the true derivative can be obtained. We will discuss more of this calculation in the following Marquardt algorithm section.

Complex Examples in Dynamic Learning

As we did in Chapter 5 for the FP algorithm, in order to demonstrate the applicability of the two basic dynamic learning models to any recurrent network configuration using the BTT algorithm, we give the following two more complex examples. These examples are the same as those described in Chapter 5. In both examples, the complexity of the problem comes from the addition of a fixed neural network that is involved in the training of the neural network controller.

In the first complex example, shown in Figure 6.4, the network to be trained is NN($\underline{w}$) and the fixed network is NN'. The configuration is a combination of the previously described basic dynamic learning Models I and II. Thus, both Equations 6.3 and 6.4 are used to calculate the STT. We start by calculating the STT of the output layer of the fixed network at time stage k,

$$\frac{\partial F}{\partial y(k)} = \frac{\partial^e F}{\partial y(k)} + \frac{\partial^e y(k+1)}{\partial y(k)} \frac{\partial F}{\partial y(k+1)} \qquad (6.12)$$

where $\frac{\partial^e F}{\partial y(k)}$ can be obtained using the basic backpropagation method. Then, the STT in the output layer of the training network at time stage k is calculated as

$$\frac{\partial F}{\partial v(k)} = \frac{\partial^e y(k)}{\partial v(k)} \frac{\partial F}{\partial y(k)} \qquad (6.13)$$

Because the derivative information from the future is already contained in $\frac{\partial F}{\partial y(k)}$ we then only need to apply the chain rule to $\frac{\partial F}{\partial v(k)}$ once. This is the same step as our calculation of the STT for each hidden layer of the training network.

The configuration for the second complex example is similar to the first one. However, a TDL component has been added between the two networks as shown in Figure 6.5. Even with this addition, this example is still a combination of the basic dynamic learning Models I and II. Thus, the STT in the output layer of the fixed network at time stage k is same as in the first complex example and is calculated as

$$\frac{\partial F}{\partial y(k)} = \frac{\partial^e F}{\partial y(k)} + \frac{\partial^e y(k+1)}{\partial y(k)} \frac{\partial F}{\partial y(k+1)}. \tag{6.14}$$

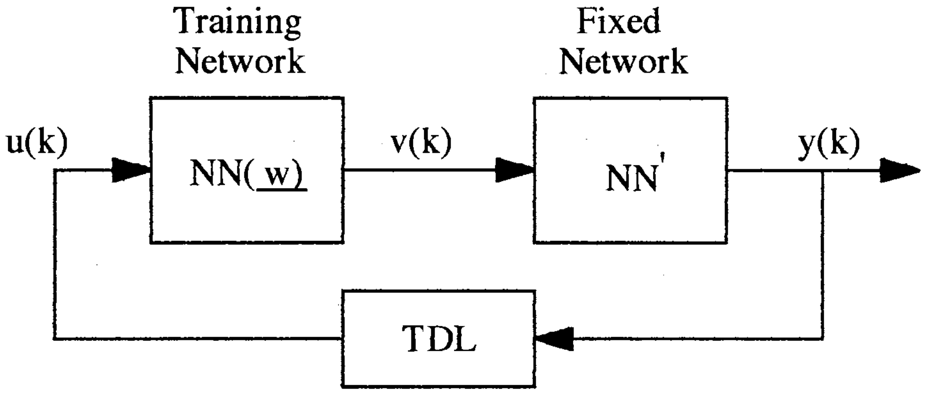


Figure 6.4 First Complex Example for Dynamic Learning

However, in this example, the following calculation of the STT in the output layer of the training network at time stage k is a little more complex than in the first example

$$\frac{\partial F}{\partial v(k)} = \frac{\partial^e F}{\partial v(k)} + \frac{\partial^e v(k+1)}{\partial v(k)} \frac{\partial F}{\partial v(k+1)} \qquad (6.15)$$

The term $\frac{\partial F}{\partial v(k)}$ in Equation 6.15 is no longer treated as the equivalent to a derivative in a hidden layer in the network. It has a direct feed-in from the future time stage. This results in the double usage of the chain rules as shown in Equations 6.1 and 6.2.

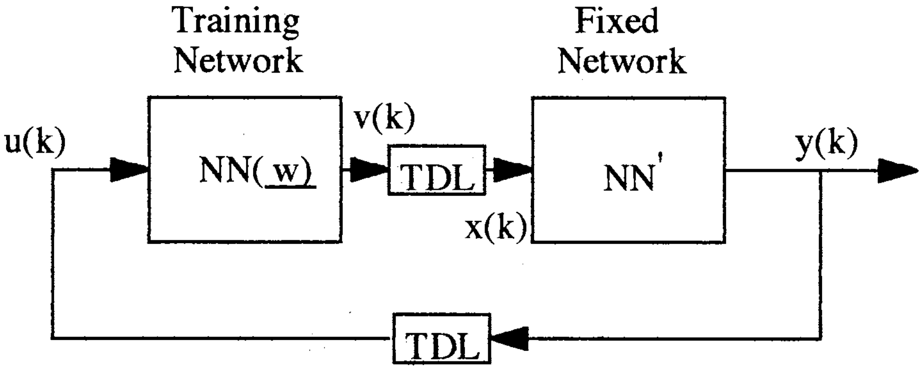


Figure 6.5 Second Complex Example for Dynamic Learning

Marquardt Optimization

We start this section by reviewing the optimization equation for the Marquardt method as described in Equation 5.21. It is rewritten as Equation 6.15

$$\Delta \underline{w} = [J(\underline{w})^T J(\underline{w}) + \mu I]^{-1} [J(\underline{w})^T \underline{f}(\underline{w})]. \qquad (6.15)$$

In order to compute the change in the network parameters, $\Delta \underline{w}$ in Equation 6.15, a Jacobian matrix $J(\underline{w})$ must first be obtained. The following Jacobian matrix is the derivative of the error function $\underline{f}$ with respect to the network parameter vector $\underline{w}$

$$J(\underline{w}) = \begin{bmatrix} \frac{\partial f_1(\underline{w})}{\partial w_1} & \frac{\partial f_1(\underline{w})}{\partial w_2} & \cdots & \frac{\partial f_1(\underline{w})}{\partial w_n} \\ \frac{\partial f_2(\underline{w})}{\partial w_1} & \frac{\partial f_2(\underline{w})}{\partial w_2} & \cdots & \frac{\partial f_2(\underline{w})}{\partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_N(\underline{w})}{\partial w_1} & \frac{\partial f_N(\underline{w})}{\partial w_2} & \cdots & \frac{\partial f_N(\underline{w})}{\partial w_n} \end{bmatrix} \qquad (6.16)$$

Using BTT in the Marquardt Method

As we stated in Chapter 5 the FP algorithm and Marquardt optimization are a perfect combination for use in dynamic learning. This will be better understood after we have finished deriving the following tedious calculation process for implementing the BTT algorithm in the Marquardt method.

First note that it is the derivative of the error function, not the derivative of the objective function, that is the key term in the Jacobian matrix described in Equation 6.16. To coordinate this particular requirement in the Marquardt optimization method using BTT, we must rewrite Equations 6.1 and 6.2, replacing the objective function $\underline{F}$ with the error function $\underline{f}$, as follows

$$\frac{\partial \underline{f}}{\partial \underline{w}} = \sum_k \frac{\partial^e a_k}{\partial \underline{w}} \frac{\partial \underline{f}}{\partial a_k} \tag{6.17}$$

and

$$\frac{\partial \underline{f}}{\partial a_k} = \frac{\partial^e \underline{f}}{\partial a_k} + \frac{\partial^e a_{k+1}}{\partial a_k} \frac{\partial \underline{f}}{\partial a_{k+1}} \tag{6.18}$$

For illustration and simplicity, only a single recurrent output is assumed in Equations 6.17 and 6.18. The error function, for a single output network in one learning epoch, is defined as

$$\underline{f} = \underline{t} - \underline{y} \tag{6.19}$$

where $\underline{t}$ and $\underline{y}$ are the desired output and the plant output respectively. Therefore, the derivative of $\underline{f}$ with respect to the parameter vector $\underline{w}$ can be expressed as

$$\frac{\partial \underline{f}}{\partial \underline{w}} = \frac{\partial(\underline{t} - \underline{y})}{\partial \underline{w}} = -\frac{\partial \underline{y}}{\partial \underline{w}} \tag{6.20}$$

It is clear now that we need to use the STT from the BTT algorithm to find the PTT at each time stage.

Converting STT to PTT

To compute the PTT at each time stage using the STT, we first must consider the fact that the present network output error is a result of both the present network inputs and past network outputs. In addition, the STT is a accumulation of the effects (to the output error) from present network inputs and future network outputs. So, for any time stage k in a learning epoch we must consider the following:

1.) At the last stage, $\frac{\partial f_k}{\partial a_k}$ is computed from Equation 6.18 with the fact that future network outputs have no effect on the current error ($\frac{\partial f_k}{\partial a_{k+1}} = 0$):

$$\begin{aligned}\frac{\partial f_k}{\partial a_k} &= \frac{\partial^e f_k}{\partial a_k} + \frac{\partial^e a_{k+1}}{\partial a_k}\frac{\partial f_k}{\partial a_{k+1}} \\ &= -1 + 0 \\ &= -1,\end{aligned} \tag{6.21}$$

2.) The effect from the past network outputs on the present output error f_k, in terms of STT is calculated as

$$
\begin{aligned}
\frac{\partial f_k}{\partial a_{k-1}} &= \frac{\partial^e f_k}{\partial a_{k-1}} + \frac{\partial^e a_k}{\partial a_{k-1}} \frac{\partial f_k}{\partial a_k} \\
&= 0 + \frac{\partial^e a_k}{\partial a_{k-1}}(-1) \\
&= -\frac{\partial^e a_k}{\partial a_{k-1}}, \\
\frac{\partial f_k}{\partial a_{k-2}} &= \frac{\partial^e f_k}{\partial a_{k-2}} + \frac{\partial^e a_{k-1}}{\partial a_{k-2}} \frac{\partial f_k}{\partial a_{k-1}} \\
&= 0 + \frac{\partial^e a_{k-1}}{\partial a_{k-2}}\left(-\frac{\partial^e a_k}{\partial a_{k-1}}\right) \\
&= -\frac{\partial^e a_{k-1}}{\partial a_{k-2}} \frac{\partial^e a_k}{\partial a_{k-1}}, \\
&\;\vdots \\
\frac{\partial f_k}{\partial a_1} &= \frac{\partial^e f_k}{\partial a_1} + \frac{\partial^e a_2}{\partial a_1} \frac{\partial f_k}{\partial a_2} \\
&= 0 + \frac{\partial^e a_2}{\partial a_1}\left(-\frac{\partial^e a_3}{\partial a_2} \cdots \frac{\partial^e a_k}{\partial a_{k-1}}\right) \qquad (6.22) \\
&= -\frac{\partial^e a_2}{\partial a_1} \frac{\partial^e a_3}{\partial a_2} \cdots \frac{\partial^e a_k}{\partial a_{k-1}}.
\end{aligned}
$$

By combining Equations 6.21 and 6.22 we have Equation 6.23. Thus, we have converted the STTs into the PTT at k-th time stage for the Marquardt optimization.

$$
\frac{\partial f_k}{\partial \underline{w}} = \sum_{q=k}^{1} \frac{\partial^e a_q}{\partial \underline{w}} \frac{\partial f_k}{\partial a_q} \tag{6.23}
$$

A Comparison of FP and BTT

Since the computation results of calculating the true derivative are the same whether using either the FP or BTT algorithms, we will compare instead the computation time (measured in CPU time) and the number of floating point operation (measured in flops) of both the FP and BTT algorithms. The same computer simulation, which will be described as Case 5 in Chapter 7, was performed for both the algorithms. Table 6.1 lists the computation times of the FP and BTT algorithms for the first ten learning epochs of the simulation.

TABLE 6.1 Comparisons of Computation Time (in CPU Time) between the FP and BTT Algorithms

Learning Epoch	CPU Time of FP	CPU Time of BTT	Ratio of CPU Time (FP/BTT)
1	3.8964	3.0736	1.2677
2	3.9297	7.2542	0.5417
3	4.9108	8.0886	0.6071
4	4.5164	7.7872	0.5800
5	4.7820	8.0839	0.5915
6	4.9714	7.9106	0.6284
7	4.6630	7.9936	0.5878
8	4.8822	8.0602	0.6057
9	4.7837	8.0717	0.5927
10	4.8083	8.0781	0.5952

The CPU time unit in the table is minutes and both the FP and the BTT simulations were performed on a Macintosh Centris 650 computer using MATLAB. Because random initial weights are used in this simulation, the first learning epoch was composed of 50 trajectories. For the second learning epoch and all epochs thereafter, the designed number of 10 trajectories were used. Therefore, the first learning epoch is excluded when averaging the FP/BTT ratio of CPU time (column 4 of Table 6.1). The average FP/BTT ratio is 0.5922 which means the FP algorithm only took about half of the CPU time that the BTT algorithm consumed.

Table 6.2 lists the number of floating point operations (in flops) for the calculation of the FP and BTT algorithms for the first ten learning epochs of the simulation. From the point of view of the floating point operation, by excluding the first learning epoch as explained above, the BTT algorithm performs a little better than the FP algorithm.

The results shown in Tables 6.1 and 6.2 are not consistent. This is due to the numbers of matrix operations executed in both the algorithms. In the FP algorithm, most of the computations are matrix operations. In contrast, for the BTT algorithm most of the calculations are scalar loop oriented. Since MATLAB is a software dedicated to matrix operation, it is no surprise that in this particular environment the FP algorithm used less CPU time than the BTT algorithm. When a general software environment is used, the BTT algorithm would have a little edge over the FP algorithm in computation time.

TABLE 6.2 Comparisons of Computation Time (in Flops) between the FP and BTT Algorithms

Learning Epoch	FP	BTT
1	12094210	2700520
2	12091280	11665910
3	12091280	11665910
4	12091280	11665910
5	12091280	11665910
6	2091354	11090893
7	12091280	11665910
8	12091280	11665910
9	12091280	11665910
10	12091280	11665910

CHAPTER VII

NEUROCONTROL USING DYNAMIC LEARNING

The purpose of this chapter is to demonstrate the training of a neural network controller using forward perturbations, the dynamic learning algorithm described in Chapter 5, in an on-line adaptive fashion. Several papers [4][10][11][12] have described the theory of FP, but we have found no successful implementation of the algorithm for model reference adaptive control reported in the literature. Therefore, it is very significant and the most important result of this research that a satisfactory implementation of forward perturbation has been reached through the computer simulations performed in this chapter.

We start with a description of the model reference adaptive control (MRAC) method, which is a commonly used architecture in adaptive control. Then, to integrate it with a neural network plant identifier, the MRAC method is reconfigured. This allows the neural network controller to be trained with the FP dynamic learning algorithm. To illustrate the gradient calculation in the dynamic learning process, a scalar version is given first. Then, a general vector version is described. Finally, to investigate the performance of the indirect MRAC using FP, extensive computer simulations ranging from simple to complicated cases were performed.

The simulations begin with a case of a linear first-order dynamic plant using a linear neural network controller. After three successively more complex cases, the simulations end with the case of a nonlinear second-order dynamic plant using a nonlinear neural network controller. As described in the last section of this chapter, the results of all the simulations were successful. Thus, this research provides us with a potential tool to deal with the more complex nonlinear dynamic applications found in the real world.

MRAC in Linear Dynamic Systems

There are two different approaches, direct and indirect, to the adaptive control of linear dynamic systems. The key difference between these two methods is the source of the information which is used to adjust the parameters of the controller. In direct adaptive control, as shown in Figure 7.1, the parameters of the controller are directly adjusted by the plant output error. In comparison, the indirect adaptive control, shown in Figure 7.2, has a plant identifier placed between the controller and the plant output error. The parameters of the controller are chosen so that the parameters of the plant identifier can represent the parameters of the true plant.

In linear adaptive control systems, such as the one shown in Figure 7.3, the input u(k) and the output $y_p(k+1)$ of the plant are connected to the TDLs so that the delayed values -- u(k-1), u(k-2), ... , u(k-n) and $y_p(k)$, $y_p(k-1)$, ... , $y_p(k-n+1)$ -- are the outputs of the TDLs. A linear combination from the controller reference input r(k) and the delayed values from both the plant input u(k) and the plant

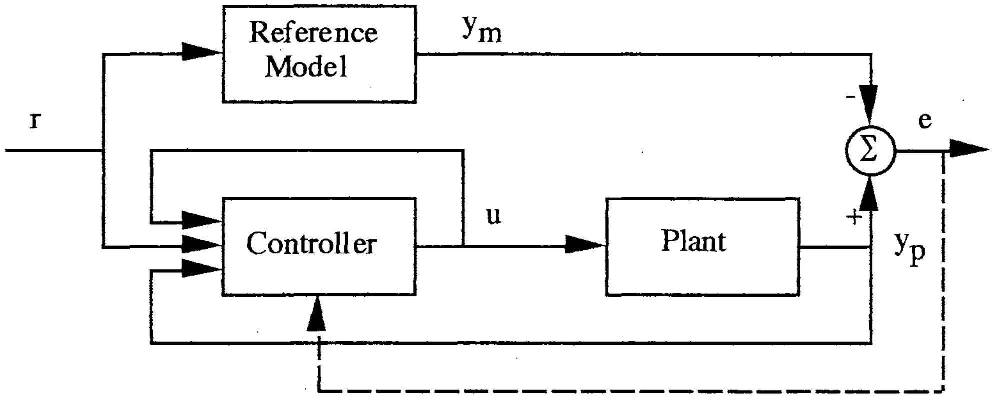


Figure 7.1 Direct Adaptive Control

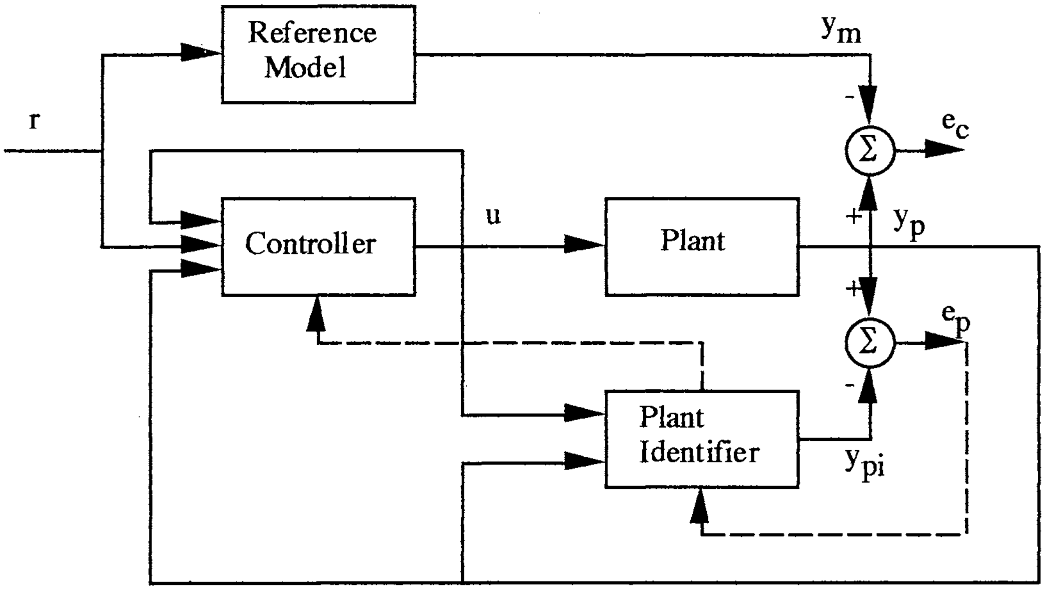


Figure 7.2 Indirect Adaptive Control

output $y_p(k+1)$ is used to produce the controller output

$$u(k) = \underline{x}^T(k)\underline{p}(k), \tag{7.1}$$

where the controller parameter vector $\underline{x}(k)$ is

$$\underline{x}(k) = [x_1(k)\ x_2(k)\ \ldots x_{2n+1}(k)]^T \tag{7.2}$$

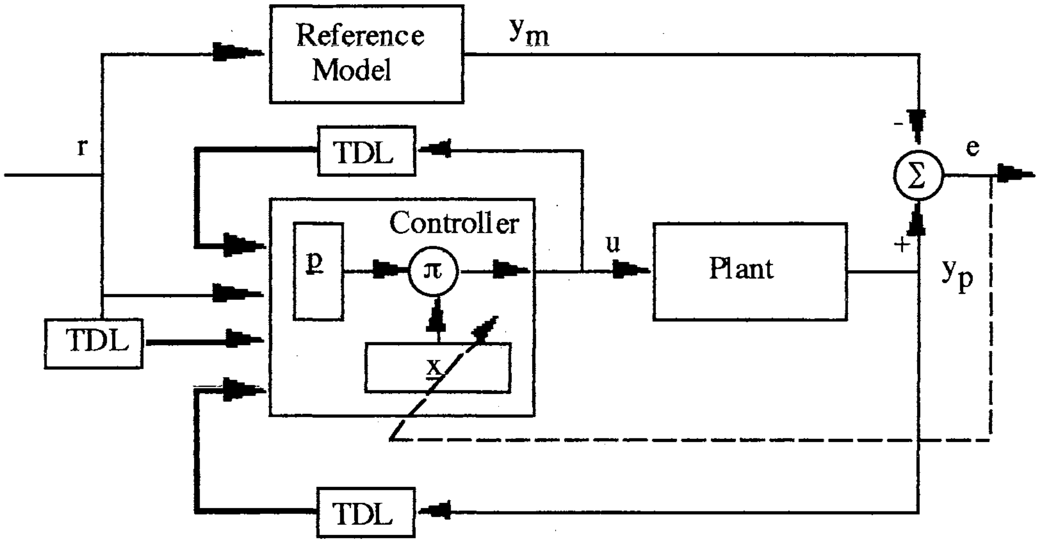


Figure 7.3 The Illustration of Adaptive Control

and the controller input vector $\underline{p}$ is

$$\begin{aligned}\underline{p}(k) = [&r(k)\ u(k-1)\ u(k-2)\ \ldots u(k-n) \\ &y_p(k)\ y_p(k-1)\ \ldots y_p(k-n+1)]^T\end{aligned} \tag{7.3}$$

For a given reference model, it can be shown that a constant parameter vector $\underline{x}_*$ exists such that the controlled plant responds exactly like the reference model when the designated plant input

$$u_*(k) = \underline{x}_*^T(k)\underline{p}(k), \tag{7.4}$$

is applied [4]. With this reference model the constant parameter vector $\underline{x}_*$ is obtained in an adaptive fashion from the plant input-output measurements. The MRAC method has been thoroughly studied and complete descriptions can be found in the book by Narendra and Annaswamy [13].

However, very little research has been reported on the use of adaptive controllers for plants described by nonlinear difference or differential equations. It is the control of such systems using the neural networks that is the primary focus of this document.

Indirect MRAC Using Neural Networks

The gradient method is used in the training of neural network controllers, and with a neural network plant identifier NN_p providing a medium for passing through the output error to the controller, it is feasible to use the backpropagation algorithm for the training of neural network controllers. The architecture of the indirect MRAC using neural networks is shown in Figure 7.4. Using this new configuration, the gradient calculations for adjusting the parameters of the neural network controller are derived in the following section.

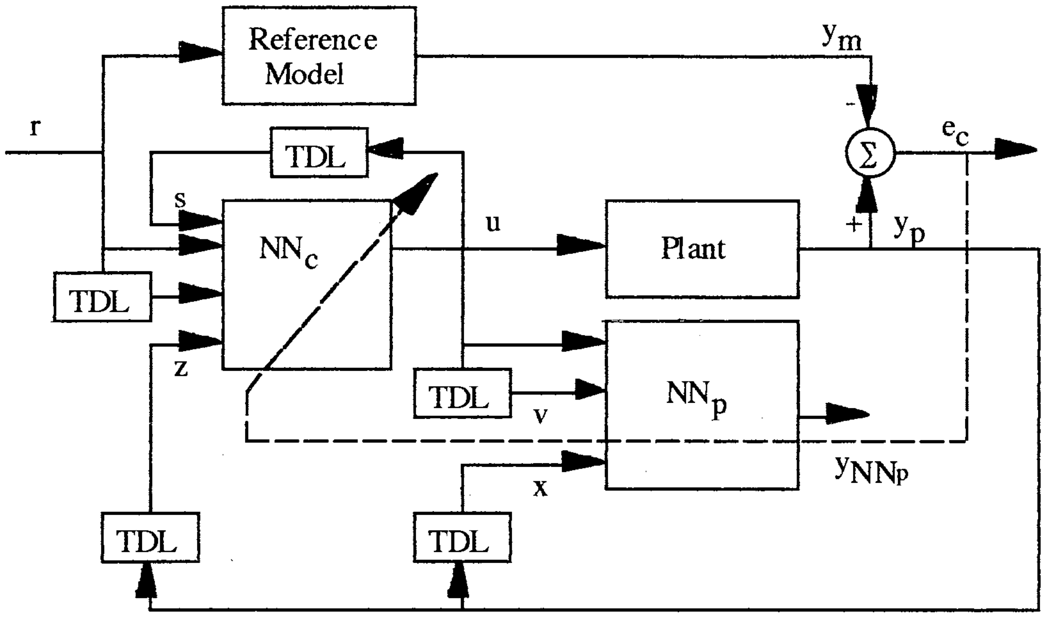


Figure 7.4 Indirect MRAC Using Neural Networks

Indirect MRAC Using Forward Perturbation

For simplicity, it is assumed that each TDL in Figure 7.4 has a single output. Since the output of the neural network identifier $y_{NN}(k+1)$ is only an estimate of the plant output $y_p(k+1)$, the plant output is chosen as the target function for the gradient calculations. From the discussions of the gradient calculations for the two complex examples in Chapter 5, it is shown that the true gradient of the plant output w.r.t. the parameter vector $\underline{w}$ of the neural network controller NN_c through time is obtained as

$$\begin{aligned}\frac{\partial y(k+1)}{\partial \underline{w}} &= \frac{\partial y^e(k+1)}{\partial \underline{w}} + (\frac{\partial v(k)}{\partial \underline{w}}\frac{\partial^e y(k+1)}{\partial v(k)} + \\ &\quad \frac{\partial u(k)}{\partial \underline{w}}\frac{\partial^e y(k+1)}{\partial u(k)} + \frac{\partial x(k)}{\partial \underline{w}}\frac{\partial^e y(k+1)}{\partial x(k)}) \\ &= 0 + (\frac{\partial u(k-1)}{\partial \underline{w}}\frac{\partial^e y(k+1)}{\partial u(k-1)} + \\ &\quad \frac{\partial u(k)}{\partial \underline{w}}\frac{\partial^e y(k+1)}{\partial u(k)} + \frac{\partial y(k-1)}{\partial \underline{w}}\frac{\partial^e y(k+1)}{\partial y(k-1)})\end{aligned} \quad , \quad (7.5)$$

where all the explicit derivative terms can be obtained by basic backpropagation. The true gradients $\frac{\partial y(k)}{\partial \underline{w}}$ and $\frac{\partial u(k-1)}{\partial \underline{w}}$ in Equation 7.5 are the perturbations propagated forward from the past time stage to the present time stage. The present true gradient $\frac{\partial u(k)}{\partial \underline{w}}$ in Equation 7.5 can be obtained as

$$\begin{aligned}\frac{\partial u(k)}{\partial \underline{w}} &= \frac{\partial u^e(k)}{\partial \underline{w}} + (\frac{\partial^e u(k)}{\partial s(k)}\frac{\partial s(k)}{\partial \underline{w}} + \frac{\partial^e u(k)}{\partial z(k)}\frac{\partial z(k)}{\partial \underline{w}}) \\ &= \frac{\partial u^e(k)}{\partial \underline{w}} + (\frac{\partial^e u(k)}{\partial u(k-1)}\frac{\partial u(k-1)}{\partial \underline{w}} + \\ &\quad \frac{\partial^e u(k)}{\partial y(k-1)}\frac{\partial y(k-1)}{\partial \underline{w}})\end{aligned} \quad , \quad (7.6)$$

where all the explicit derivative terms can also be obtained by basic backpropagation. This concludes the gradient calculations for the indirect adaptive neural network MRAC method using the FP dynamic learning algorithm.

The generalized vector versions of Equations 7.5 and 7.6 are given in Equations 7.7 and 7.8.

$$\begin{aligned}
\frac{\partial y(k+1)}{\partial \underline{w}} &= \frac{\partial y^e(k+1)}{\partial \underline{w}} + \frac{\partial \underline{v}(k)^T}{\partial \underline{w}} \frac{\partial^e y(k+1)}{\partial \underline{v}(k)} + \\
&\quad \frac{\partial u(k)}{\partial \underline{w}} \frac{\partial^e y(k+1)}{\partial u(k)} + \frac{\partial \underline{x}(k)^T}{\partial \underline{w}} \frac{\partial^e y(k+1)}{\partial \underline{x}(k)} \\
&= 0 + \sum_i \frac{\partial u(k-i)}{\partial \underline{w}} \frac{\partial^e y(k+1)}{\partial u(k-1)} + \\
&\quad \frac{\partial u(k)}{\partial \underline{w}} \frac{\partial^e y(k+1)}{\partial u(k)} + \\
&\quad \sum_j \frac{\partial y(k-j+1)}{\partial \underline{w}} \frac{\partial^e y(k+1)}{\partial y(k-j+1)},
\end{aligned} \tag{7.7}$$

$$\begin{aligned}
\frac{\partial u(k)}{\partial \underline{w}} &= \frac{\partial u^e(k)}{\partial \underline{w}} + \frac{\partial \underline{s}(k)^T}{\partial \underline{w}} \frac{\partial^e u(k)}{\partial \underline{s}(k)} + \frac{\partial \underline{z}(k)^T}{\partial \underline{w}} \frac{\partial^e u(k)}{\partial \underline{z}(k)} \\
&= \frac{\partial u^e(k)}{\partial \underline{w}} + \sum_i \frac{\partial u(k-i)}{\partial \underline{w}} \frac{\partial^e u(k)}{\partial u(k-1)} + \\
&\quad \sum_j \frac{\partial y(k-j+1)}{\partial \underline{w}} \frac{\partial^e u(k)}{\partial y(k-j+1)},
\end{aligned} \tag{7.8}$$

Computer Simulations

The computer simulations performed for implementation of the indirect MRAC using the FP dynamic learning algorithm were categorized into the following five cases,

(1) First-order linear plant with linear controller

(2) First-order linear plant with nonlinear controller

(3) Second-order linear plant with linear controller

(4) Second-order linear plant with nonlinear controller

(5) Second-order nonlinear plant with nonlinear controller

The general learning steps in the training procedure, which apply to all the above cases, are,

Step 1: The training data sets were obtained on-line by stacking the data points that resulted from several runs, up to a preset total number. A run started with a randomly chosen initial condition and was given a random constant reference. It stopped when either the output was out of the output space, or a preset maximum number of data points was reached. Thus, a run is equivalent to a plant trajectory.

Step 2: The true gradients of the plant output w.r.t. the network parameters at each time stage of a learning epoch were also computed when the training data was collected.

Step 3: Unlike the on-line executed training data collected in Step 1 and the on-line gradients calculations performed in Step 2, the optimization process in this step, using the Marquardt method, was performed off-line. The reason is, as described in Chapter 2, that some number of iterations have to be performed in order to find the appropriate value of μ in the Marquardt optimization. Thus, instead of the real physical plant, the neural network plant identifier is used in this step.

Step 4: The training was stopped when either a maximum learning epoch number or a least mean squared error goal was reached.

In the discussion of each computer simulation, presented first is a learning curve of the neural network controller that was obtained after the training. Then, for the linear controllers, the network parameters are compared to the true coefficients of the corresponding linear transfer functions to check if they match. To evaluate the trained nonlinear controllers, several tests were performed to compare the network controlled plant output with the desired output of the reference model. A parallel test method, in which the reference model and the controlled plant ran independently of each other, was used in all the evaluation tests. Compared to the series-parallel test method used in static learning in Chapter 4, the parallel test method is a higher standard evaluation. To ensure that the trained nonlinear neural network controller can be operated from anywhere in the input and output spaces, the initial conditions of the tests were randomly chosen from the corresponding input spaces.

Simulation 1: First-order Linear Plant with Linear Controller

The linear neural network controller in this simulation is a multi-input, no offset, perceptron with a linear activation output. The transfer function of the reference model is given as

$$\frac{y_d}{r} = \frac{1-0.2Z^{-1}}{Z-0.4} \tag{7.9}$$

where r is the reference input and y_d is the desired output. The transfer function of the first-order plant was chosen as

$$\frac{y_p}{u} = \frac{1 - 0.3Z^{-1}}{Z - 0.5} \tag{7.10}$$

where u is the control input to the plant and y_p is the plant output. The goal of the control is to match the plant output with the desired output from the reference model. Thus, the following control input was obtained from Equations 7.9 and 7.10

$$\begin{aligned} u(k) &= y_p(k+1) - 0.5y_p(k) + 0.3u(k-1) \\ &= (0.4y_p(k) + r(k) - 0.2r(k-1)) \\ &\quad - 0.5y_p(k) + 0.3u(k-1) \\ &= 0.3u(k-1) + r(k) - 0.2r(k-1) - 0.1y_p(k) \end{aligned} \tag{7.11}$$

From Equation 7.11, one can see that the perceptron has to be a 4-input neuron. The learning curve, shown in Figure 7.5, is obtained after 45 learning epochs. The final trained neuron weights exactly matched with the coefficients from Equation 7.11. The converging process of the weights is recorded in Table 7.1.

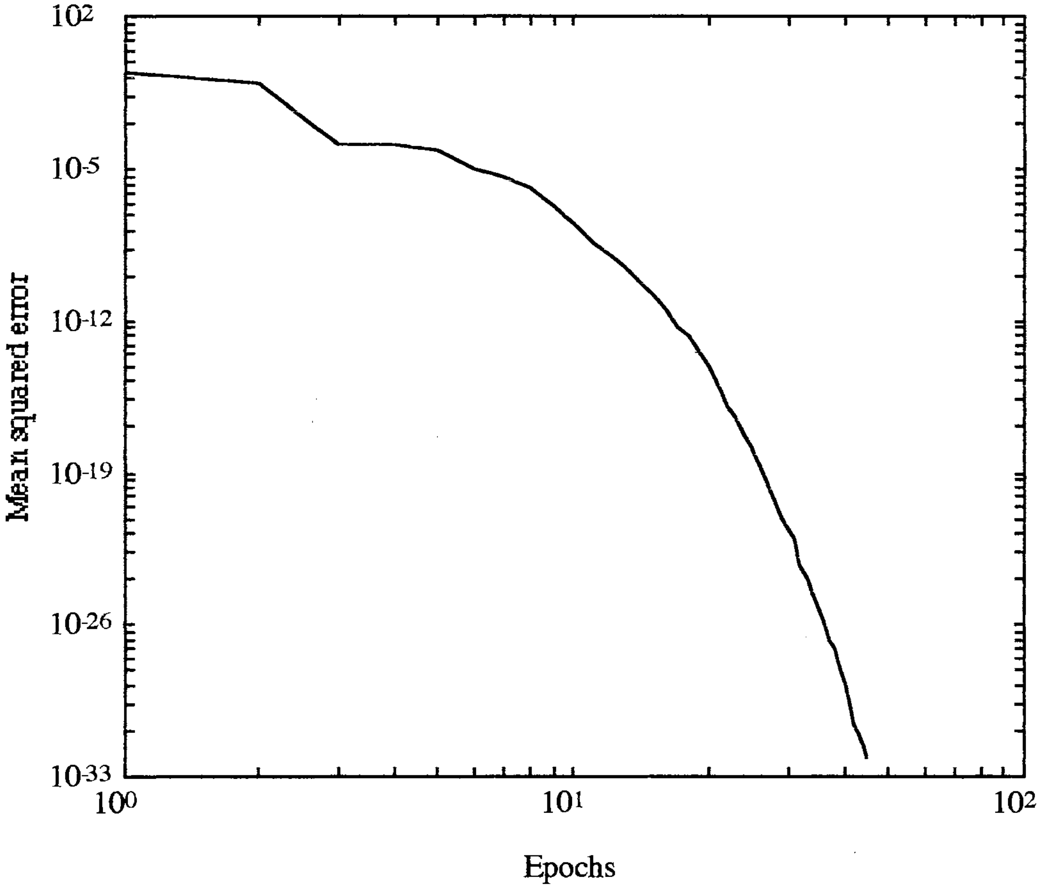


Figure 7.5 The Learning Curve from Simulation 1

TABLE 7.1 The Weights Converging Process for Simulation 1

Learning Epoch	Weights			
1	0.6009	0.9949	0.2533	-0.1236
11	0.2978	0.9999	-0.1976	-0.1002
21	0.2999	1.0000	-0.1999	-0.1000
31	0.2999	0.9999	-0.1999	-0.1000
41	0.2999	1.0000	-0.1999	-0.1000
45	0.3000	1.0000	-0.2000	-0.1000
True Coefficients	0.3000	1.0000	-0.2000	-0.1000

Simulation 2: First-order Linear Plant with Nonlinear Controller

From empirical experience, a 4-2-1 multilayer neural network was chosen for the nonlinear controller in this simulation. The learning curve, shown in Figure 7.6, was obtained after a given number of maximum learning epochs was reached. Because of the nonlinearity of the controller, we cannot verify the trained weights by comparing them directly to the coefficients in Equation 7.11 as we did in Simulation 1. Instead, four evaluation tests, shown in Figures 7.7 and 7.8, were performed for the trained network parameters. In all of the evaluation test figures in this chapter, the dashed line represents the plant and the solid line represents the reference model. As observed from Figures 7.7 and 7.8, the tracking and the regulating abilities of the trained neurocontroller, shown during the transient and the steady states respectively, are excellent in matching to the reference model.

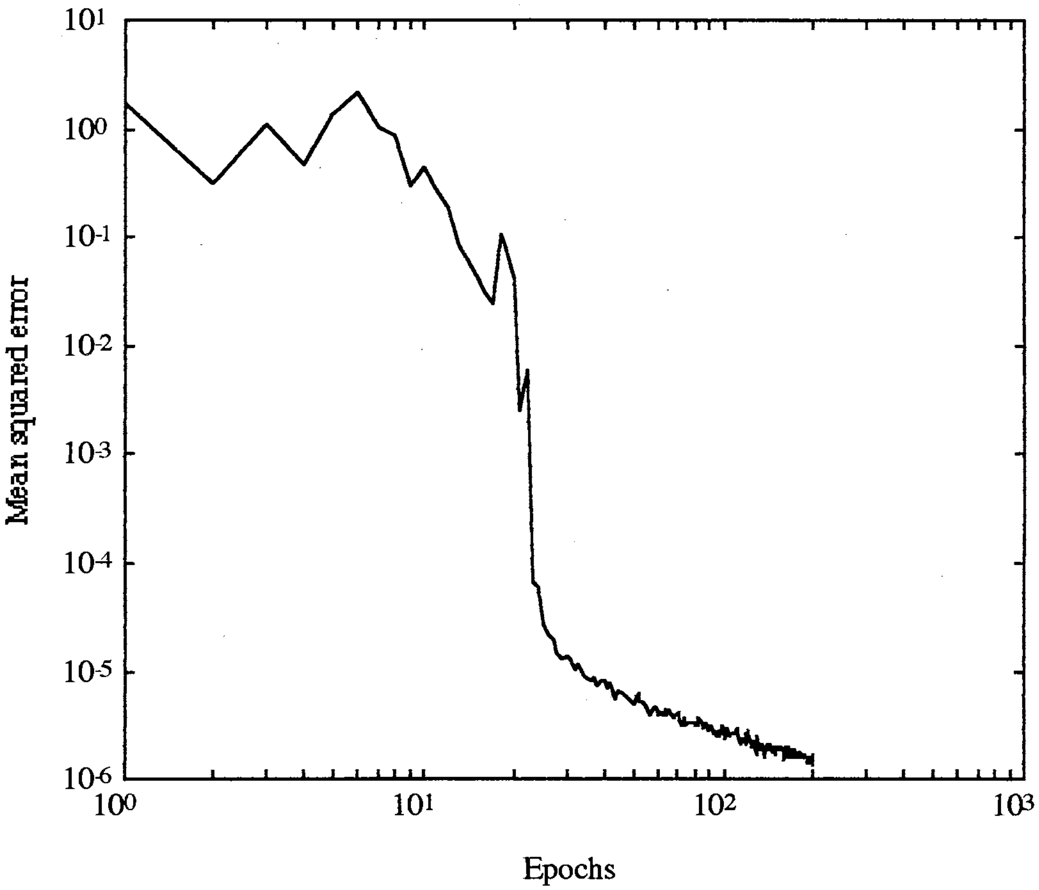


Figure 7.6 The Learning Curve from Simulation 2

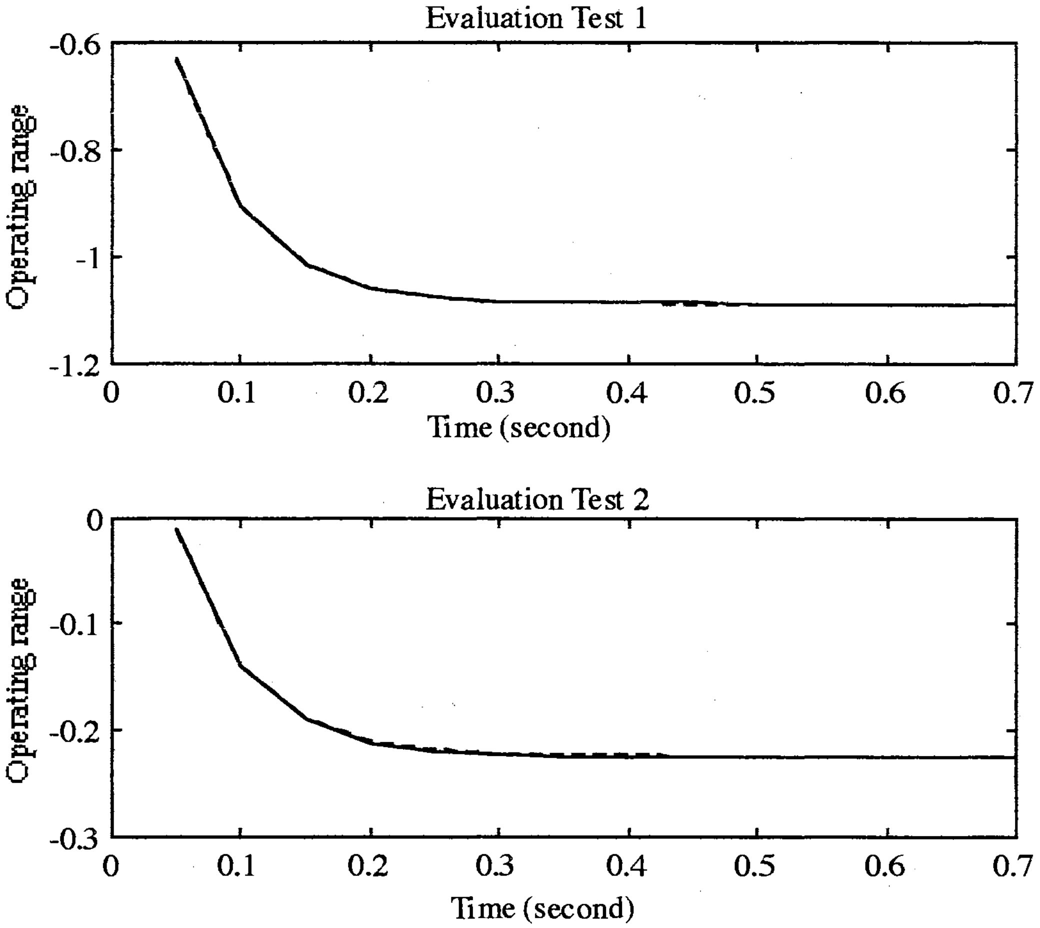


Figure 7.7 The Evaluation Tests for Simulation 2 (#1)

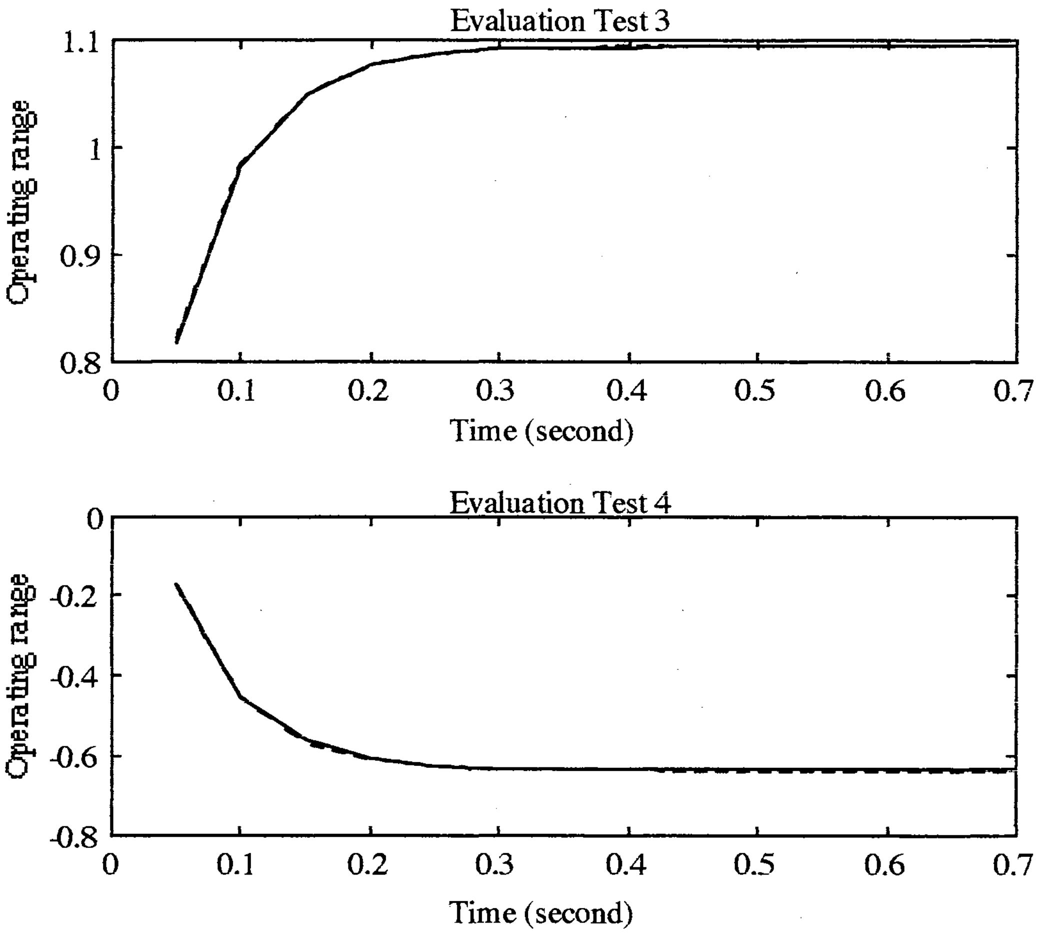


Figure 7.8 The Evaluation Tests for Simulation 2 (#2)

Simulation 3: Second-order Linear Plant with Linear Controller

The second-order reference model and the second-order plant in this simulation are described by Equations 7.12 and 7.13 respectively

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -9x_1 - 6x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 9r \end{bmatrix} \tag{7.12}$$

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -4x_1 - 4x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 4u \end{bmatrix} \tag{7.13}$$

The corresponding discrete transfer functions are, respectively,

$$y_d(k+1) = \underline{a}_d \begin{bmatrix} r(k) \\ r(k-1) \end{bmatrix} + \underline{b}_d \begin{bmatrix} y_d(k) \\ y_d(k-1) \end{bmatrix}$$

where $\underline{a}_d = [.0102 \;\; .0092]$, $\underline{b}_d = [1.1721 \;\; -.7408]$

and

$$y_p(k+1) = \underline{a}_p \begin{bmatrix} u(k) \\ u(k-1) \end{bmatrix} + \underline{b}_p \begin{bmatrix} y_p(k) \\ y_p(k-1) \end{bmatrix} \quad . \tag{7.14}$$

where $\underline{a}_p = [.0047 \;\; .0044]$, $\underline{b}_p = [1.8097 \;\; -.8187]$

To find the control input u(k), we apply the same manipulation performed in Simulation 1 to Equation 7.14 to get

$$u(k) = \{-a_p(2)u(k-1) + \underline{a_d}\begin{bmatrix} r(k) \\ r(k-1) \end{bmatrix} + (\underline{b_d} - \underline{b_p})\begin{bmatrix} y_d(k) \\ y_d(k-1) \end{bmatrix}\} / a_p(1)$$

$$= -.9355u(k-1) + [2.1770 \;\; 1.9698]\begin{bmatrix} r(k) \\ r(k-1) \end{bmatrix} + [18.8634 \;\; -16.6521]\begin{bmatrix} y_d(k) \\ y_d(k-1) \end{bmatrix}. \tag{7.15}$$

A five-input, no offset perceptron with a linear activation output was the linear neural network controller chosen for this simulation. This is a similar situation to the one in Simulation 1, but the basic training method used in Simulation 1 would not produce successful training results here. Instead the author proposed the trajectory crossing training (TCT) method and the trajectory length increasing (TLI) method that will be discussed in Chapter 9. The learning curve for this simulation is shown in Figure 7.9. The convergence process of the network parameters and the increasing trajectory length (numbers of data points or time stages ahead in one trajectory) as the learning epoch went forward were both recorded in Table 7.2. As one can see from Table 7.2, the final trained weights in Simulation 3 exactly match the true coefficients calculated from Equation 7.15.

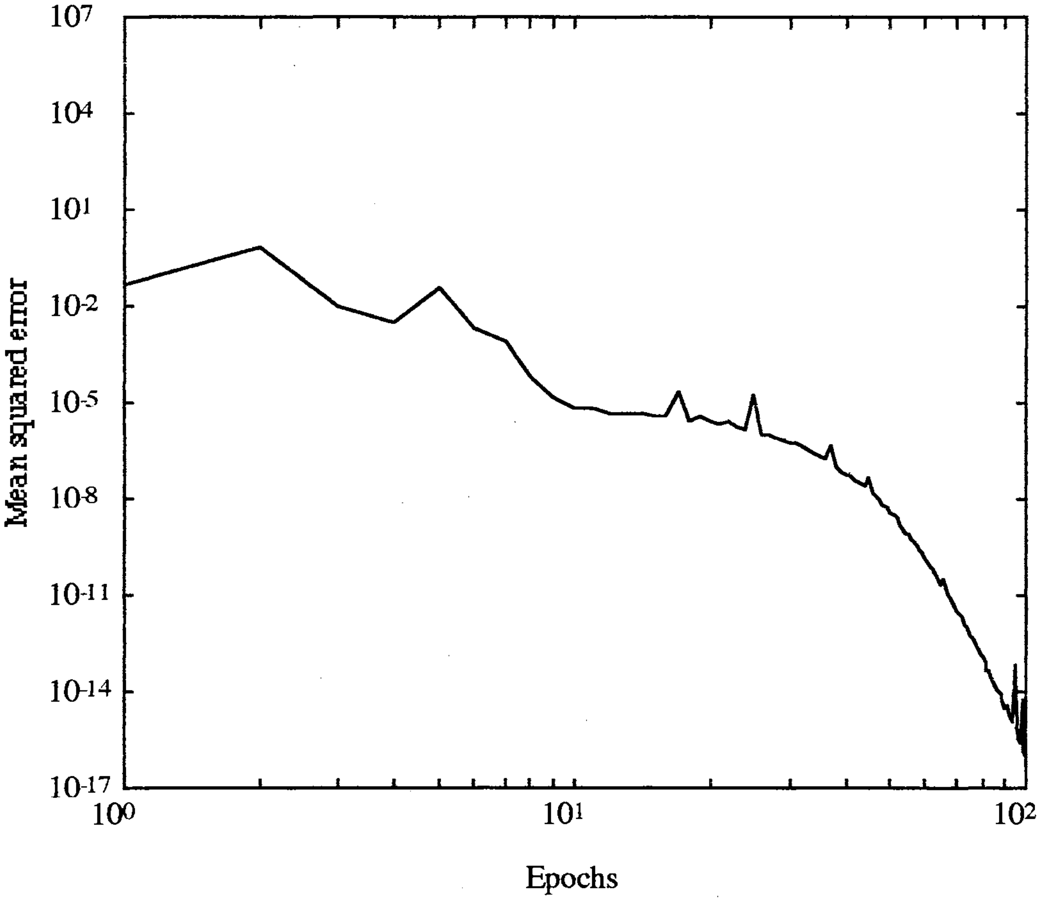


Figure 7.9 The Learning Curve for Simulation 3

TABLE 7.2 The Weights Converging Process for Simulation 3

Epoch	Data Points	Weights				
1	100	0.7045	-0.1392	0.8803	-0.6494	0.3537
11	100	-0.8931	1.9437	1.3877	19.2375	-17.5878
21	200	-0.9058	2.0407	1.6177	19.1188	-17.2922
31	400	-0.9221	2.1239	1.8289	18.9832	-16.9580
41	600	-0.9307	2.1639	1.9312	18.9050	-16.7567
51	800	-0.9344	2.1746	1.9621	18.8733	-16.6769
101	800	-0.9355	2.1770	1.9698	18.8634	-16.6521
True Coefficients		-0.9355	2.1770	1.9698	18.8634	-16.6521

Simulation 4: Second-order Linear Plant with Nonlinear Controller

From empirical experience, a 5-4-1 multilayer neural network was chosen for the nonlinear controller in this simulation. The learning curve, as shown in Figure 7.10, has about the same final mean square error as in Figure 7.6 from Simulation 2. This is due to the nonlinear neurocontroller that was used in both simulations. Another similar phenomenon shared between Figures 7.6 and 7.10 is the presence of vibrations exhibited during the declining of the learning curve. The results of the trained network controller evaluation tests are shown in Figures 7.11 and 7.12. Aided by the TCT and TLI methods, the performance of the trained controller for the second-order plant is as good as the one trained in Simulation 2 for a first-order plant.

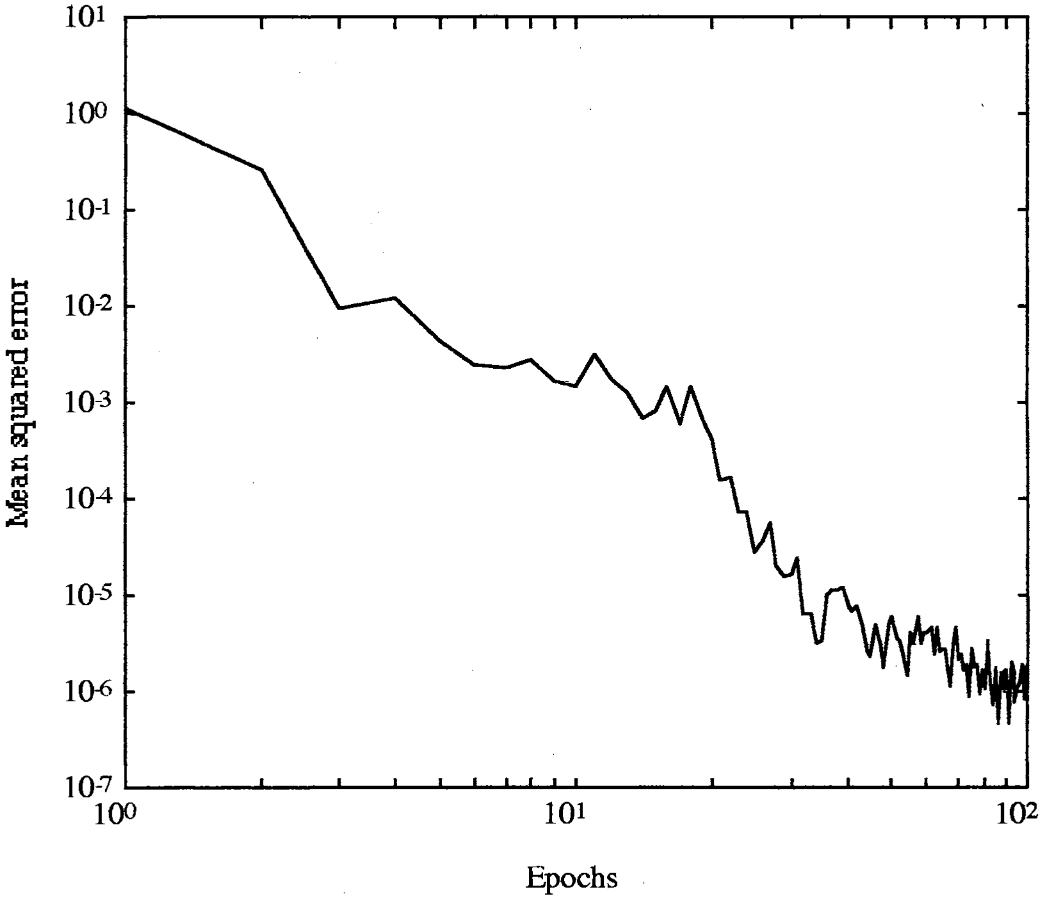


Figure 7.10 The Learning Curve for Simulation 4

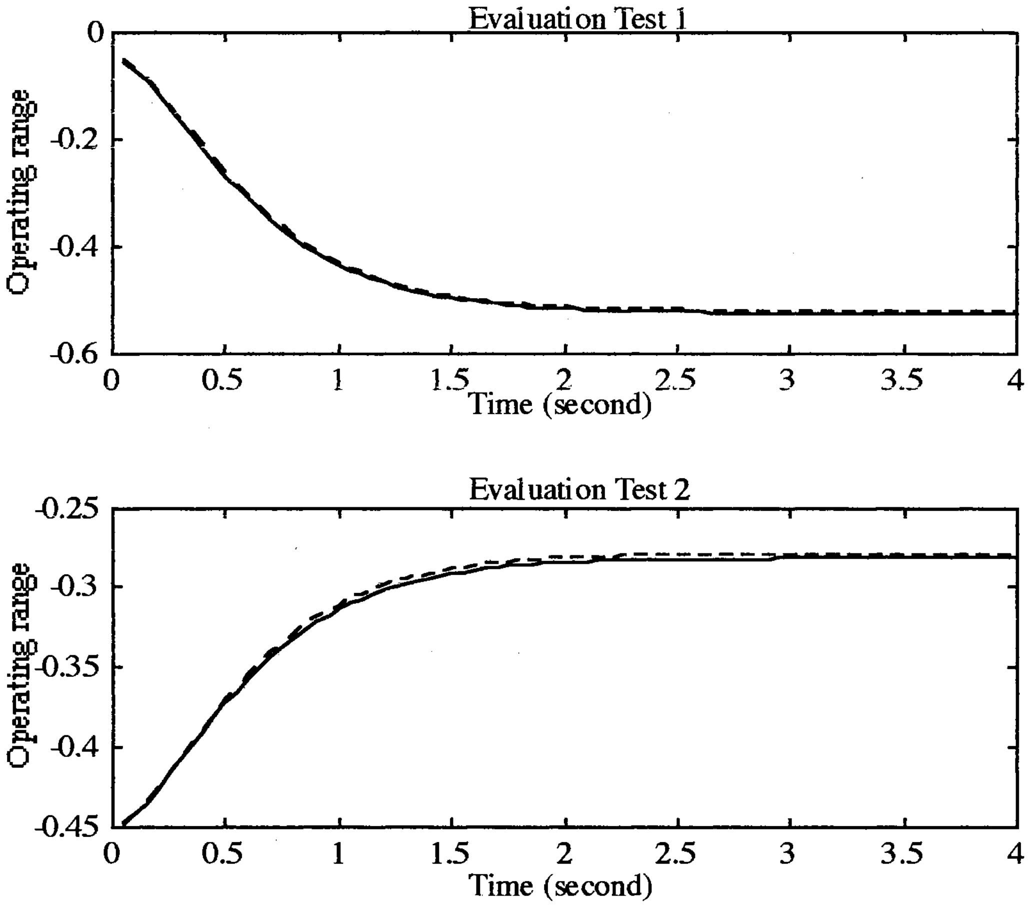


Figure 7.11 The Evaluation Tests of Simulation 4 (#1)

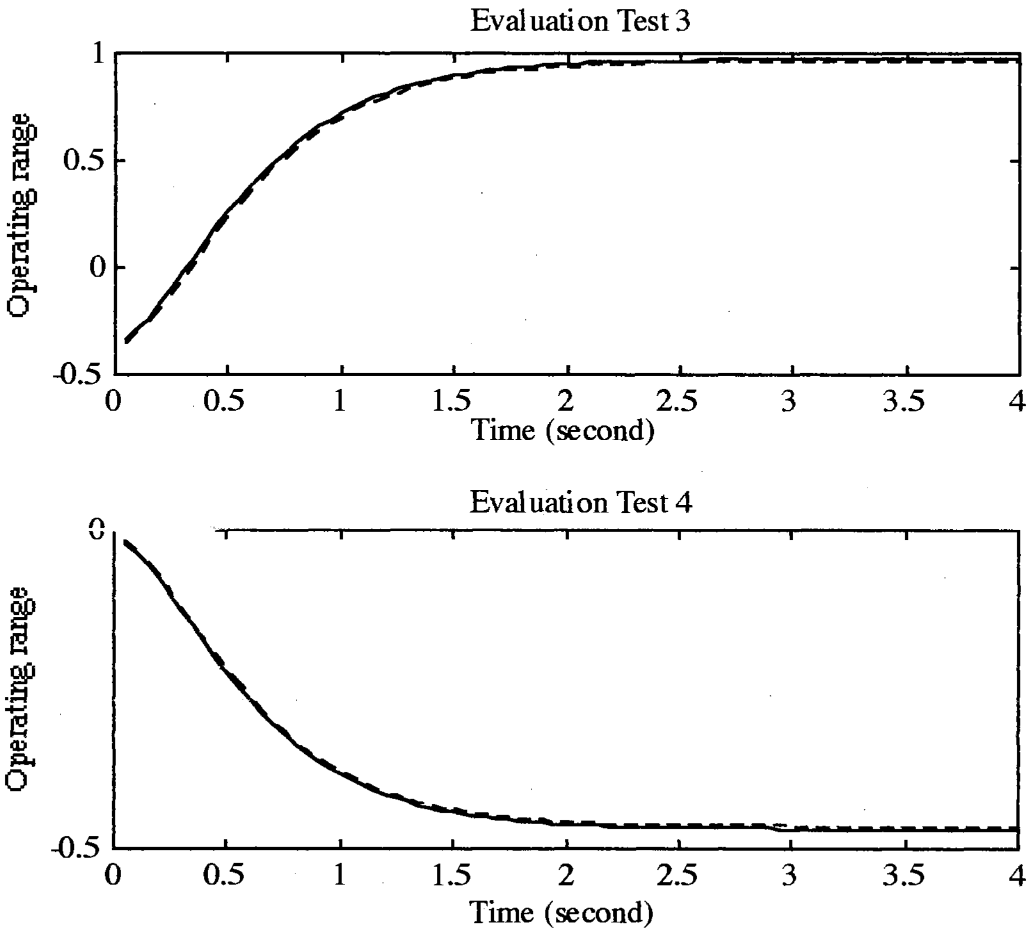


Figure 7.12 The Evaluation Tests of Simulation 4 (#2)

Simulation 5: Second-order Nonlinear Plant with Nonlinear Controller

Because this simulation contains a second-order nonlinear plant, it is the most difficult one to train compared to all the previous simulations. The swinging pendulum system described in Chapter 5 is the choice for the second-order nonlinear plant. In the previous simulations, because of the linearity of the plant, all the plant identifiers used were simply the plant model equations. However, due to the nonlinear character in the pendulum system, we will instead use the neural network pendulum identifier trained in Chapter 4.

Following the steps of the general learning procedure described in the beginning of this section, a training data set, composed of 10 trajectories with 66 data points in each trajectory, was obtained on-line for each learning epoch. Again, by empirical experience, a 5-13-1 multilayer recurrent neural network was chosen for the controller. To demonstrate the powerful training ability of the dynamic learning method using the Marquardt optimization technique, a comparison of the neural network controller responses before and after two learning epochs, is shown in Figure 7.13. It seems that, after only two learning epochs, the network controller already possesses most of the desired characteristics. This is illustrated in Figure 7.14 by the closely matching tracking effort made before 0.5 seconds and the tendency towards steady state that is exhibited toward the end of the response session.

The learning curve for the nonlinear controller, shown in Figure 7.15, was obtained after the simulation reached a preset number of maximum learning epochs. The result of the final learning epoch and the first evaluation test are presented in Figure 7.16. To verify the generalization of the trained neural network controller over the entire system operating range (or the input-output space), four more evaluation tests were also performed. The results of these tests are shown in Figure 7.17. It can be observed, from Figures 7.16 and 7.17, that the training outcome in this simulation may not be quite as accurate as the outcomes in Simulations 1 to 4. However, considering the fact that this is the first known successful simulation using dynamic learning on a second-order nonlinear plant, it is still the most significant and important result in this research.

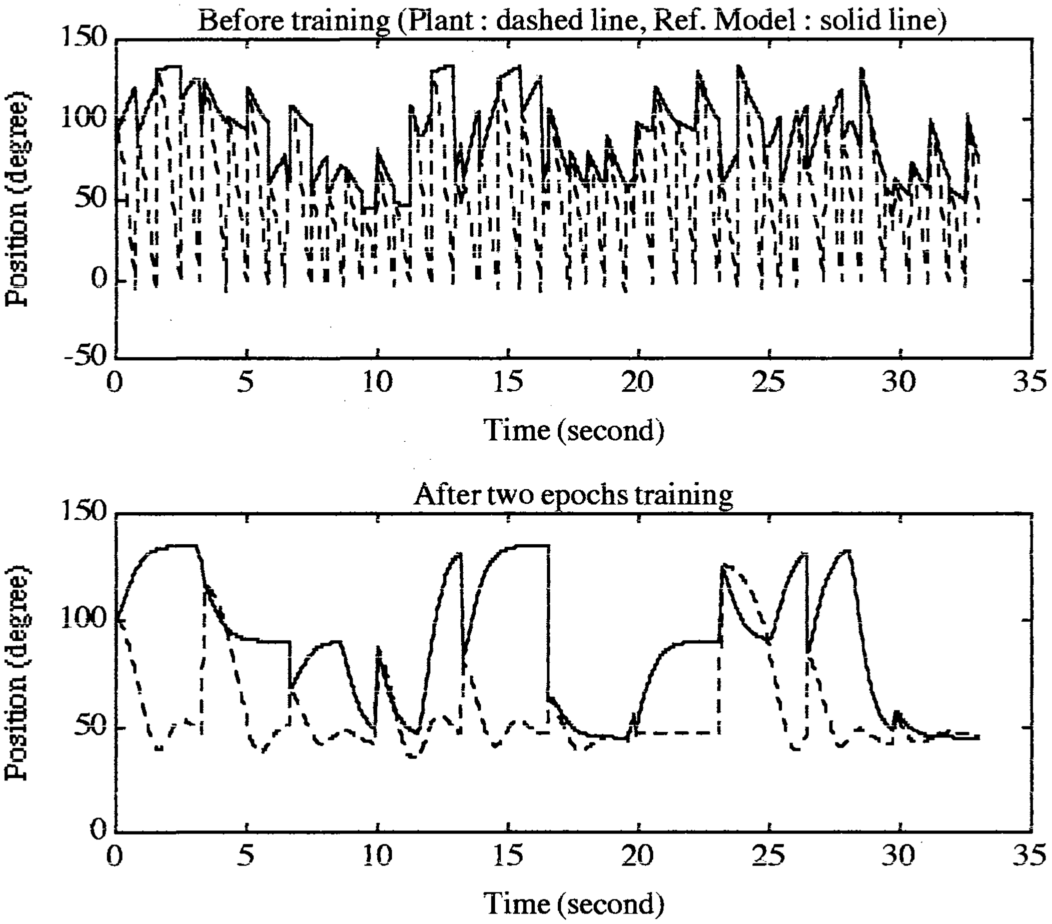


Figure 7.13 The Comparison of the Non-trained and Trained Responses of the NN_c (5-13-1)

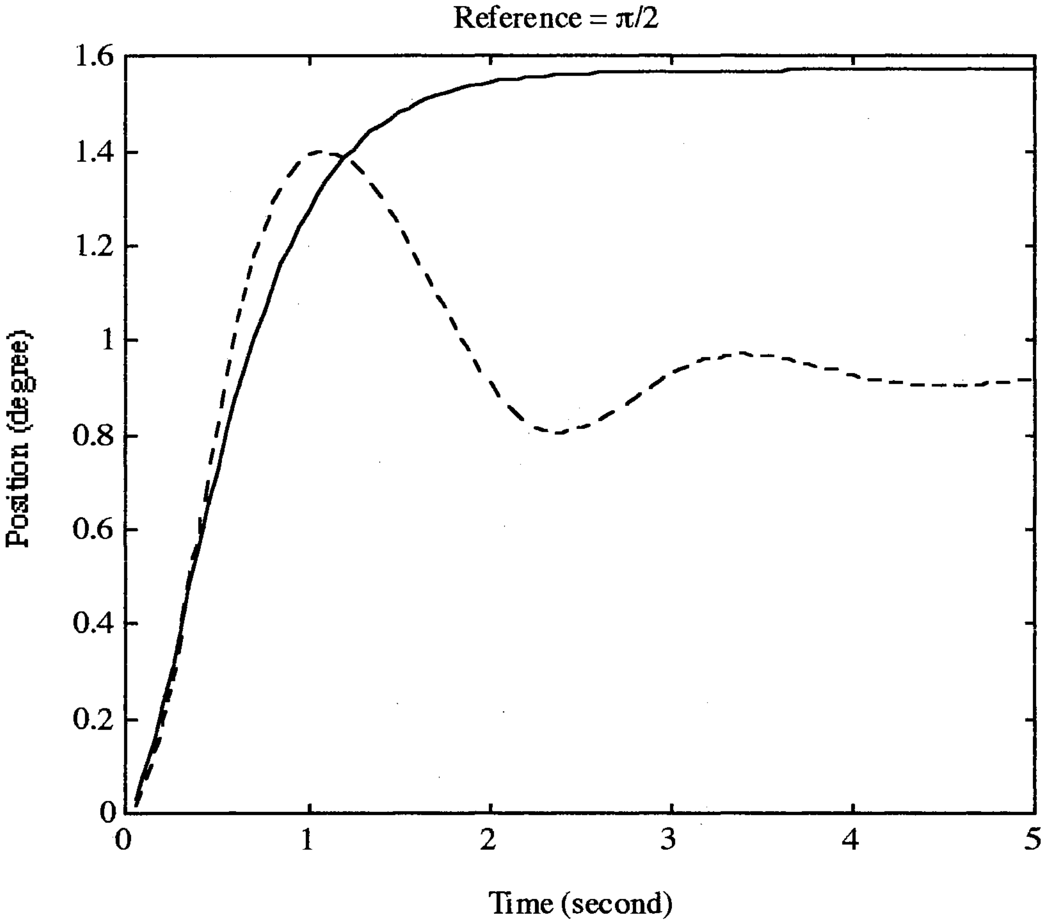


Figure 7.14 The Evaluation Test of NN_c after Two Learning Epochs

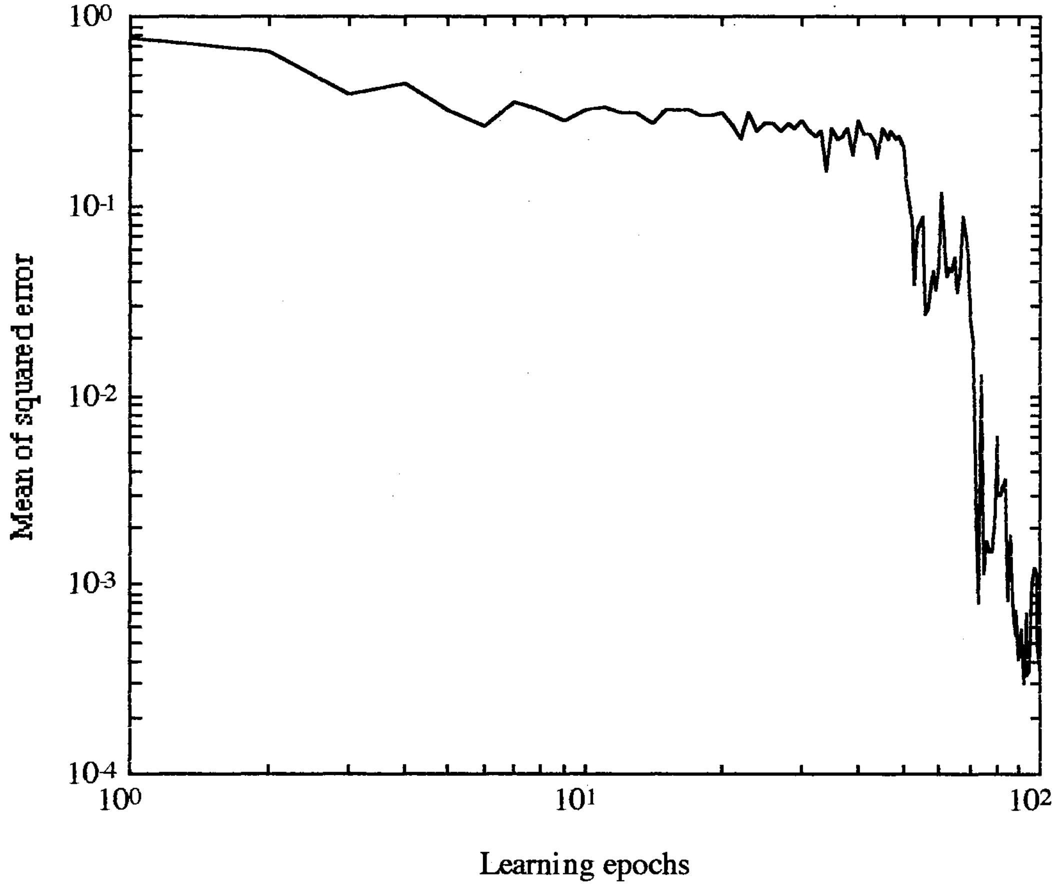


Figure 7.15 The Learning Curve for Simulation 5

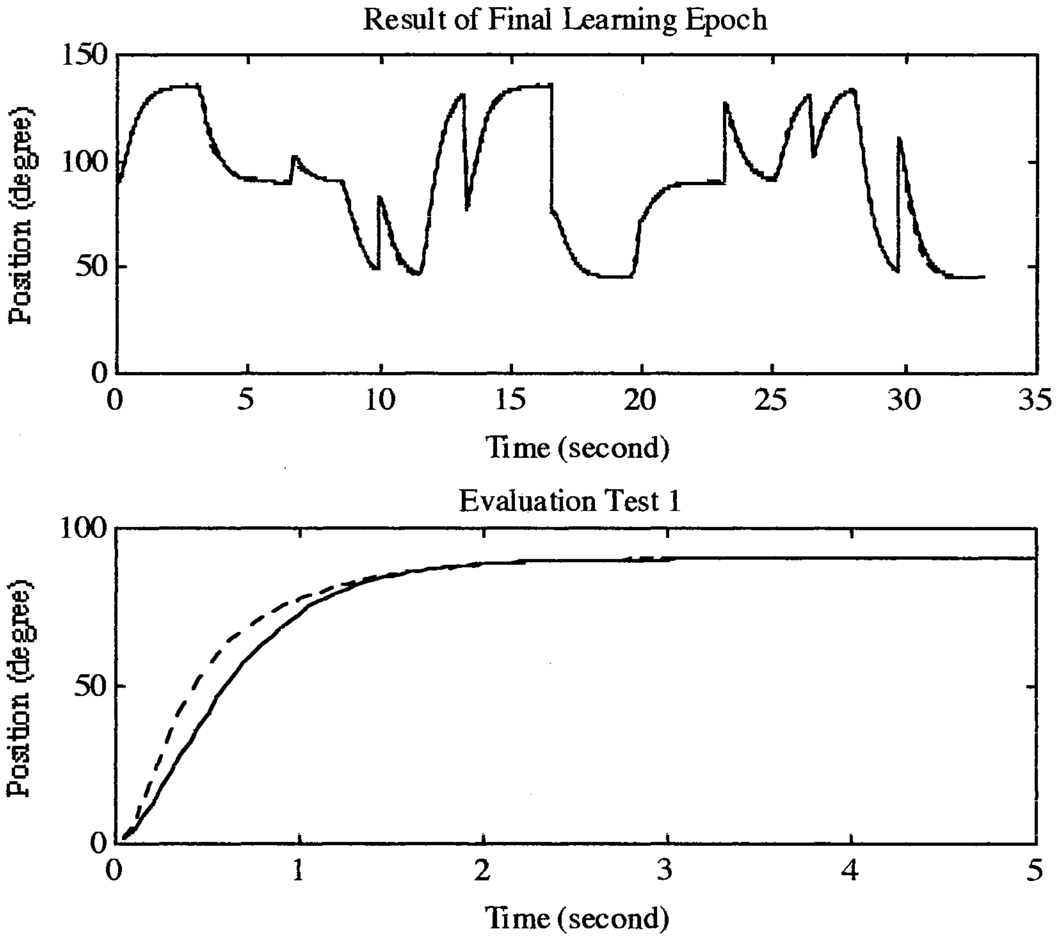


Figure 7.16 The Final Learning Epoch Result and the First Evaluation Test for Simulation 5

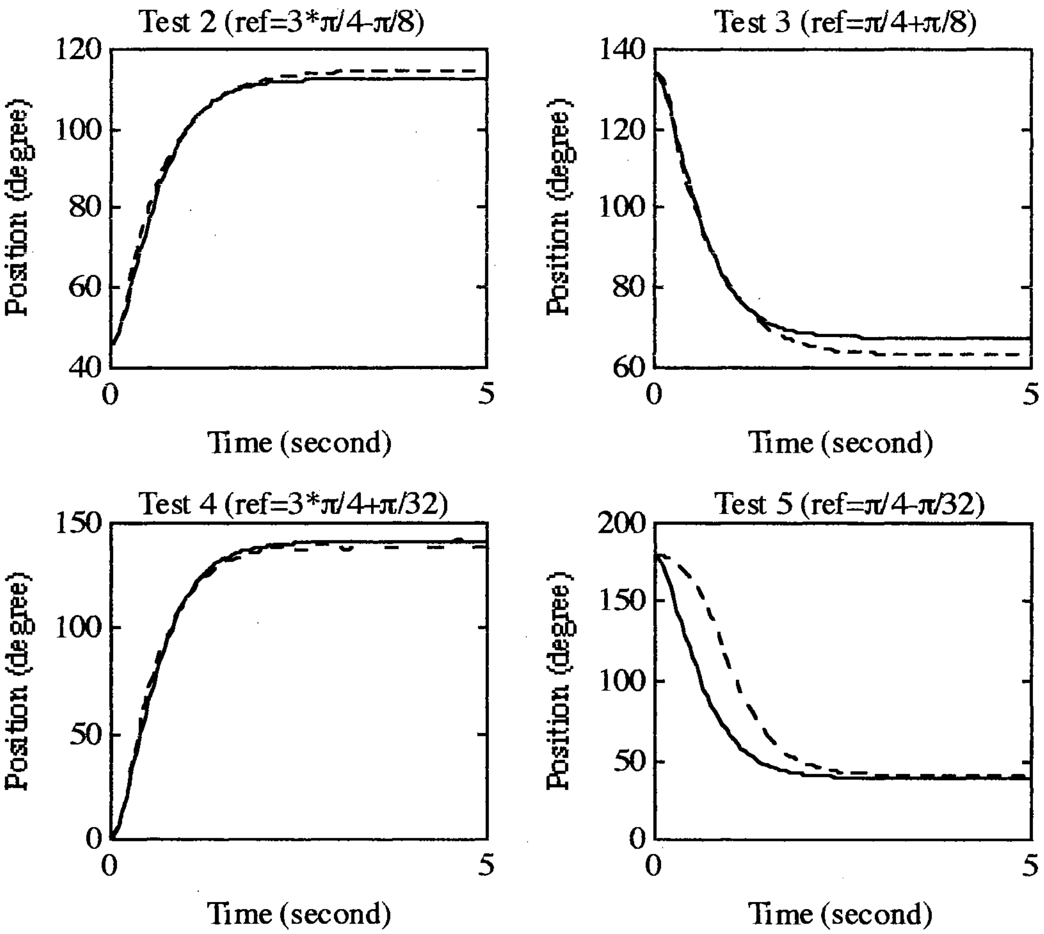


Figure 7.17 Four More Evaluation Tests for Simulation 5

Summary

Chapters 5, 6 and this chapter represent the main focus of this document. The proposed theory of dynamic learning and the forward perturbation algorithm and the backpropagation-through-time algorithm associated with it were described in Chapter 5. This chapter presented the implementation of the indirect MRAC method using the forward perturbation algorithm in five extensive computer simulations. It begins with the derivation of the gradient calculation equations for the on-line operation of the indirect MRAC using neural networks. Then, starting with the simplest case of a linear plant and a linear controller, the five dynamic learning simulations are presented, ending with the most complex case of a nonlinear plant and a nonlinear controller. All the trained controllers were evaluated using the parallel test method, which is a more precise measure than the series-parallel test method used in static learning. The outcomes of all the five simulations were satisfactory. This success was due not only to the correctness of the FP algorithm, but also to the perfectly matched Marquardt optimization method and aid from the secondary training methods.

CHAPTER VIII

REAL-TIME EXPERIMENTAL RESULTS

We reported the first successful MRAC dynamic learning computer simulation in Chapter 7. In this chapter, we will advance to implement the trained networks that use both static learning and dynamic learning on real physical systems in real-time. We start with a description of the real-time application -- the physical swinging pendulum system. Then, feedback linearization, which is described in Chapter 4, is used to illustrate the real-time control using static learning. The forward modeling of the physical pendulum system using static learning is also discussed. The forward model of the plant will be used in real-time control. Just as in computer simulation, we use a MRAC control scheme and dynamic learning to train the neurocontroller off-line. Then, we implement the trained controller in a real physical system to demonstrate real-time control using dynamic learning.

The Real Physical System Description

The real physical swinging pendulum system has three components;
(1) The physical system,

(2) The VME backplane computer system,

(3) The work station.

The functional diagram of this system is shown in Figure 8.1.

The physical system has a motor and a pendulum attached to its shaft. The sensor device for measuring motor shaft position is composed of a shaft encoder which attaches to the motor and a counter which resides on the interface board of the VME backplane computer system, as explained in the following.

There are three boards in the VME backplane computer. The first one is the interface board which contains a counter, a D/A converter and an address decoder. The counter decodes the signals from the shaft encoder. The D/A converter converts digital signals to analog signals which are sent to the motor amplifier to drive the motor. The address decoder receives read or write commands from the real-time control program. It reads the motor shaft counter as motor position input. It writes the digitized motor voltage to the D/A converter. Each of the other two boards has a Motorola 68040 CPU. The first 68040 board is the master. It has an ethernet port and a real-time operating system called Vx Works. The master serves as a communication bridge between the work station and the real-time software. The second 68040 board is the slave. It executes the real-time software, which is composed of a real-time executive and a real-time control program, to control the pendulum system.

Finally, the work station is an environment for development of real-time software, control of real-time software and graphic display of real-time test results.

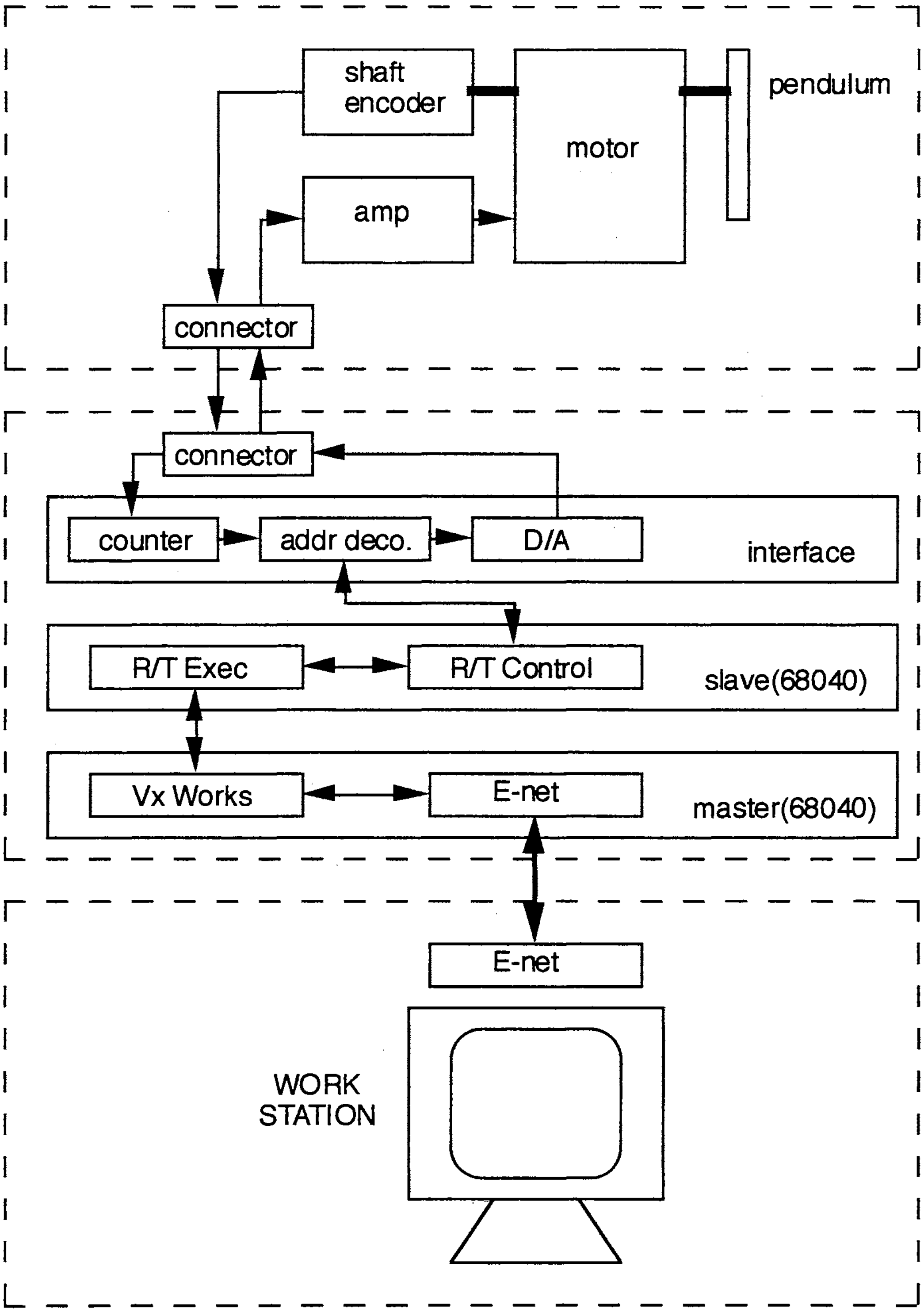


Figure 8.1 The Functional Diagram of the Pendulum System

Real-Time Control using Static Learning

We start with a review of feedback linearization, which was described in Chapter 4. Then, the procedure for training the neural network in a real-time environment is presented. The trained results include the learning curve and off-line tests of the trained network. Finally, the results from real-time control using the trained neural network in a real physical system are described.

Feedback Linearization

Since the dynamics of the physical pendulum system are not known a priore, we can only describe the plant with the following state variable equation

$$\underline{x}(k+1) = \underline{x}(k) + \Delta t \begin{bmatrix} x_2(k) \\ f(\underline{x}(k)) + (c)(u(k)) \end{bmatrix} \tag{8.1}$$

where f is the unknown nonlinearity and c is the unknown constant of the control input. We need, not only to train a neural network NN_f which can identify the nonlinearity f, but also to find out the value of the constant c. A different reference model from the one described in Equation 4.20 is selected. The discrete representation of the reference model for the real-time application is

$$\underline{x}(k+1) = \underline{x}(k) + \Delta t \begin{bmatrix} x_2(k) \\ -64x_1(k) - 16x_2(k) + 64r(k) \end{bmatrix} \tag{8.2}$$

Thus, to control the pendulum system in such way that it responds to the desired dynamics in real-time, the input will be

$$u_{FL}(k) = \frac{x_2(k) + \Delta t(-64x_1(k) - 16x_2(k) + 64r(k)) - NN_f}{(c)(\Delta t)} \quad (8.3)$$

Procedure for Training and Real-Time Control

(1) Collect the pendulum position x_1 and velocity x_2 in real-time by applying motor voltage u as described in the computer simulation in Chapter 4, a base voltage is established first, then the threshold voltage will apply to the motor when the pendulum is going out of bounds.

(2) After a total of 1,000 data points were collected, we made two training data sets with 400 points in each (no data overlapping). The static learning process with the ATS method was performed. As shown in Figure 8.2, a 2-8-1_33 neural network was selected to identify the system nonlinearity, and a neuron with only one weight (no offset) is to be trained to find the constant of the control input.

(3) Before we install the trained network in a real-time control program, we can check the network in the same way we did in the computer simulation in Chapter 4. We alternately set one of the two inputs to zero and varying the other one to get the network outputs. A half cycle sinusoidal wave and a straight line are expected in the test results.

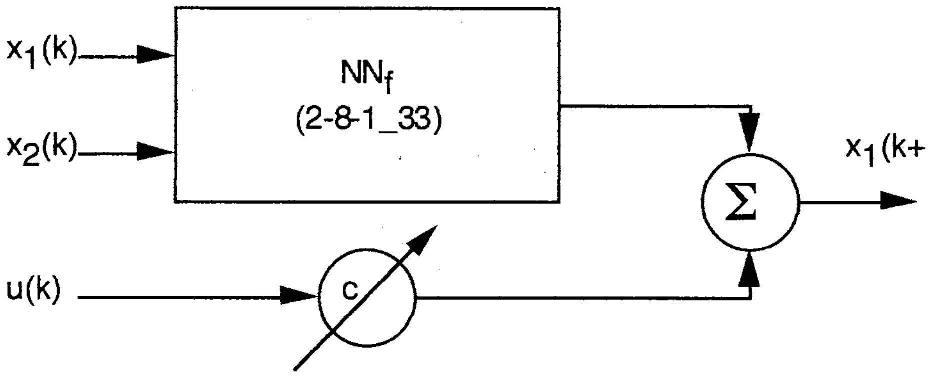


Figure 8.2 The Network Configuration to Train the NN_f and Control Input Constant C

(4) If the test results from the last step are satisfactory, we can then implement the trained network in real-time software. The control equation is based on Equation 8.3. In order to examine the performance, the desired output from the reference model is also generated in real-time.

(5) To verify the generality of the trained network NN_f and control input constant c, we perform two pendulum moving-up motions and two moving-down motions.

The Training and Real-Time Control Results

The learning curve of the NN_f and constant c is shown in Figure 8.3. After 100 epochs, the curve has already reached a minimum. In Figure 8.4, we compared the gravity nonlinearity and motor shaft friction of the physical pendulum system to the expected sinusoidal wave and straight line respectively. In the figure, the expected results and the trained network outputs are represented by solid lines and dashed lines respectively. We observed in Figure 8.4 that our trained network is quite accurate.

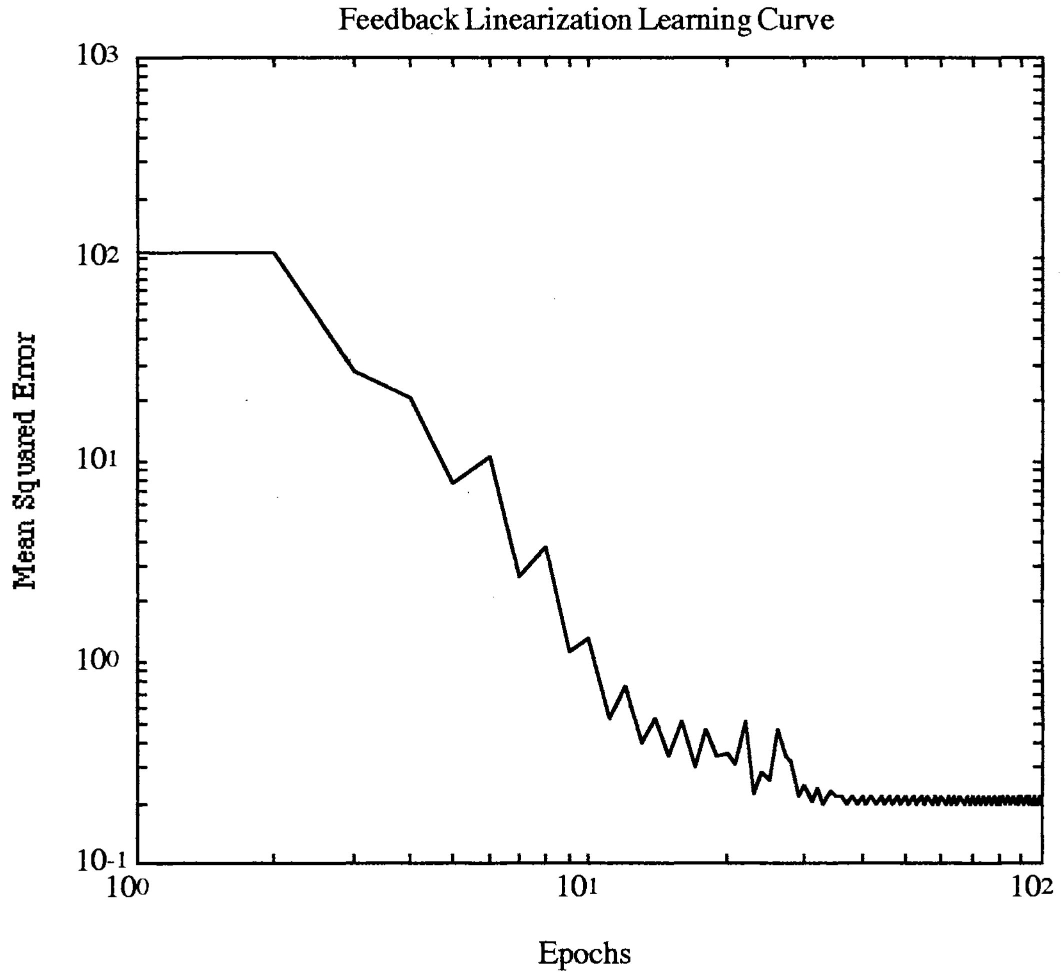


Figure 8.3 The Learning Curve of Network NN_f

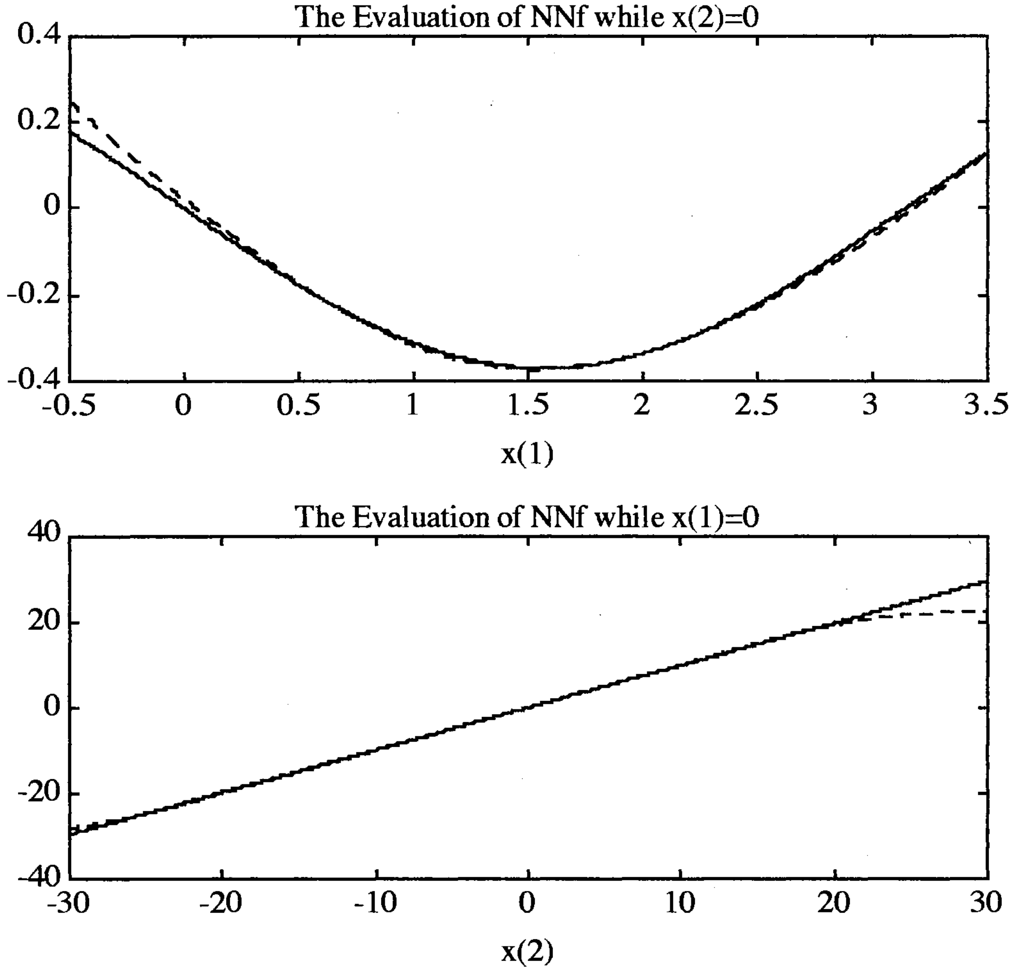


Figure 8.4 The tests of the Nonlinearity and Friction of the Physical Pendulum System

As described in the procedure, we then implement the network NN_f and control input constant c in the control equation (Equation 8.3) of the real-time software. We start the pendulum initially at a straight down position ($\theta = 0^{o}$) and set the final reference positions at horizontal (θ = 90°) and straight up (θ = 180°). The results are shown in Figure 8.5. Then, two more moving-down motions (from θ = 180° to θ = 90° and θ = 0°) were performed as shown in Figure 8.6. In both figures, the controlled plant outputs are compared to the desired outputs from the reference model.

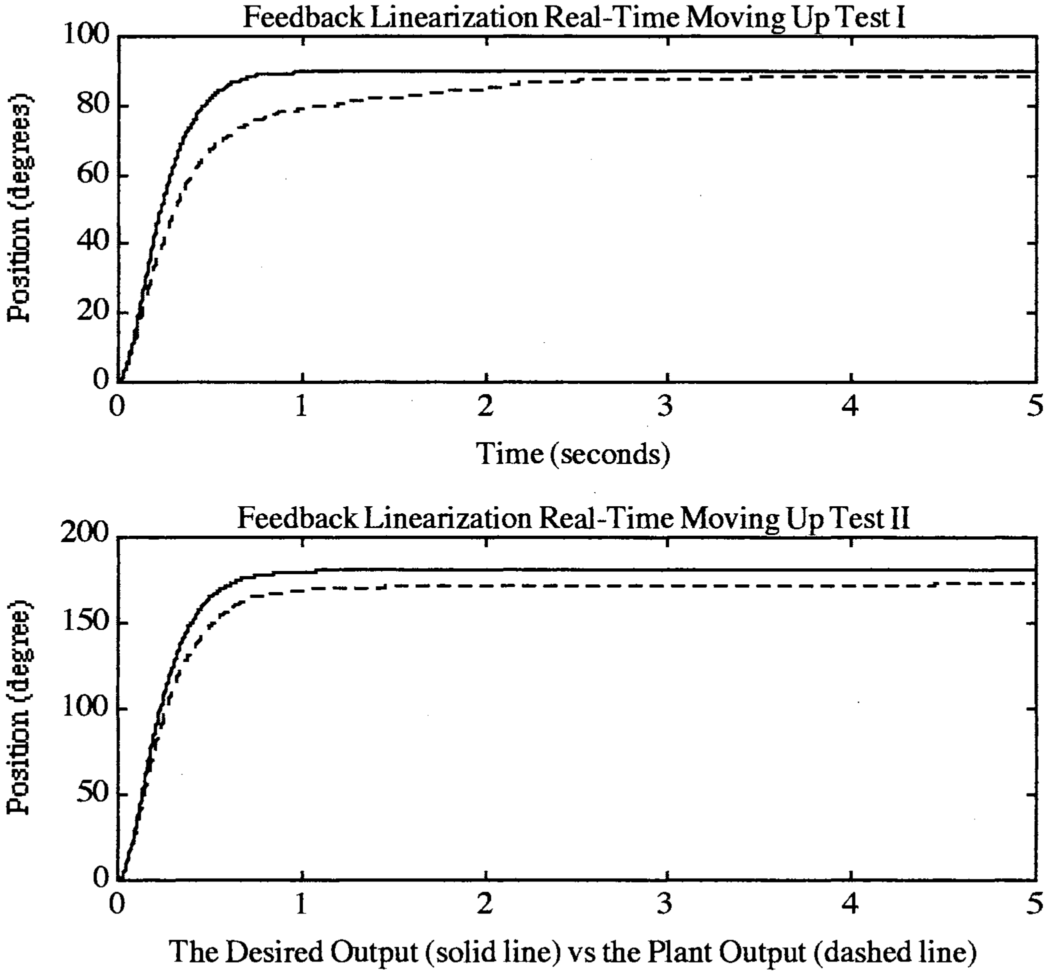


Figure 8.5 The Tests of Moving Pendulum Up in Real-Time

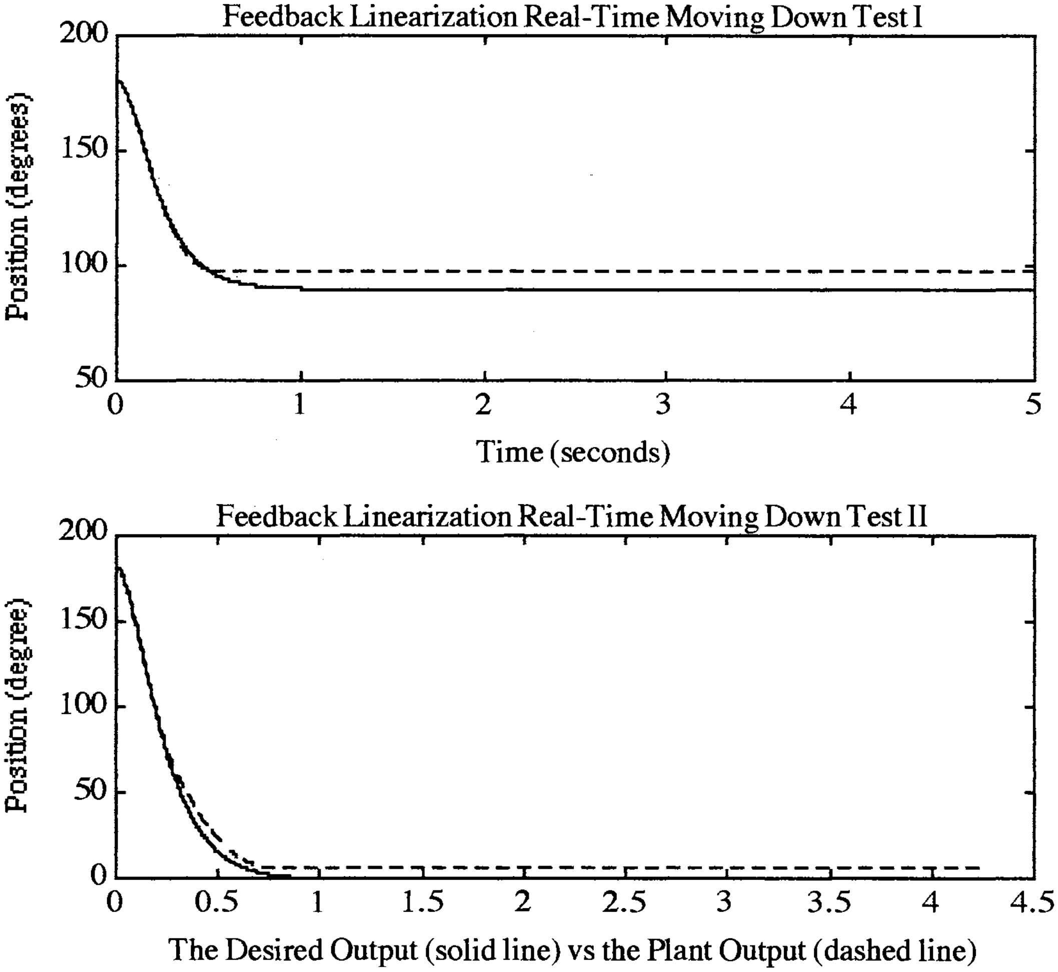


Figure 8.6 The Tests of Moving Pendulum Down in Real-Time

From Figures 8.5 and 8.6, we observed that the regulating ability of the neurocontroller is less accurate than the tracking ability. In general, we conclude that the neural network controller using the feedback linearization technique on the physical pendulum system in real-time is successful.

Real Physical System Identification

The main reason to perform forward modeling (system identification) of the physical pendulum system is because of its use in training the MRAC controller. We will describe the use of MRAC in real-time control in the next section. In this section, we start by reviewing the plant modeling process and the methods used to evaluate the trained neural network. Then, the procedures for training and real-time evaluation of the forward model of the real physical system are described. Finally, comparisons of the real physical plant and the neural network model are given.

Forward Modeling and Evaluation Methods

As shown in Figure 8.7, a feedforward neural network is selected for identifying the physical pendulum system. The inputs of the network consist of the current and previous motor control voltages u(k) and u(k-1) and the current and previous pendulum positions y(k) and y(k-1). The output of the network is the next pendulum position y(k+1).

Both the series-parallel and parallel methods are used in evaluating the plant model in real-time. The series-parallel test method, as shown in Figure 8.8, has delayed actual plant outputs as network inputs. In contrast, as shown in Figure 8.9, the network output feeds back to its input through a TDL for the parallel test method.

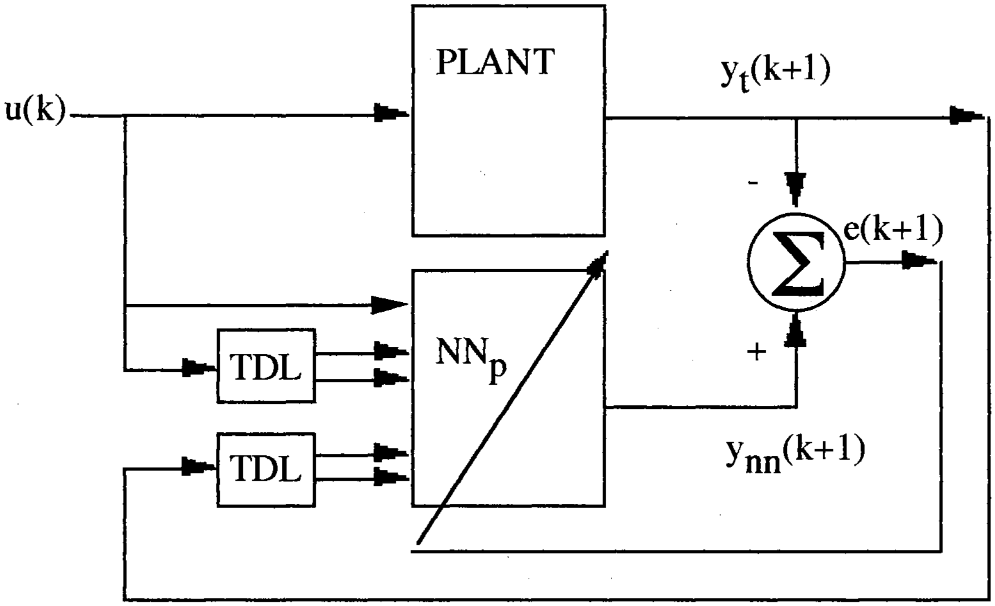


Figure 8.7 System Identification of a Real Physical Plant

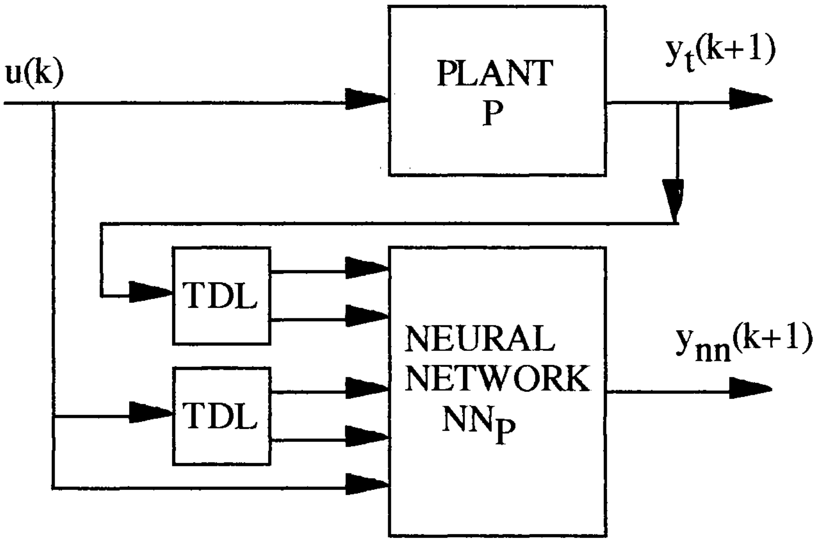


Figure 8.8 The Series-Parallel Test Method

It is obvious that the parallel test method is more difficult. If the plant model does not exactly match the real plant, then small errors in each time stage will accumulate to become a large error in the end. Contrary to this, in the series-parallel test, the true pendulum position always feeds into the plant model at each time stage. Thus there is no accumulation of error. That is why we usually have almost perfect series-parallel test results but only have approximate parallel test results.

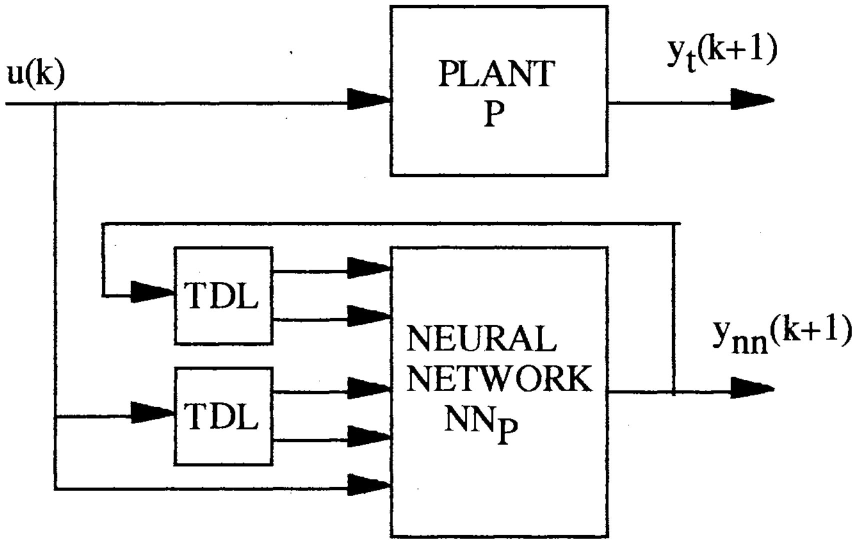


Figure 8.9 The Parallel Test Method

Procedure for Training and Real-Time Evaluation

(1) The same data collected for training in the feedback linearization can be used to train the neural network model. Only the pendulum position and the motor voltage are used in the training.

(2) A 4-5-1_31 network is selected for the plant model. Two data sets are produced for training the network with the ATS method.

(3) After the training, we implement the network model in the real-time program. Then we performed series-parallel and parallel tests to evaluate the network model.

The Training and Real-Time Evaluation Results

The learning curve of the plant model NN_p is shown in Figure 8.10. After 100 epochs, the curve has already reached a minimum. As described in the procedure, we then implement the network NN_p in the real-time software running along with the real physical system.

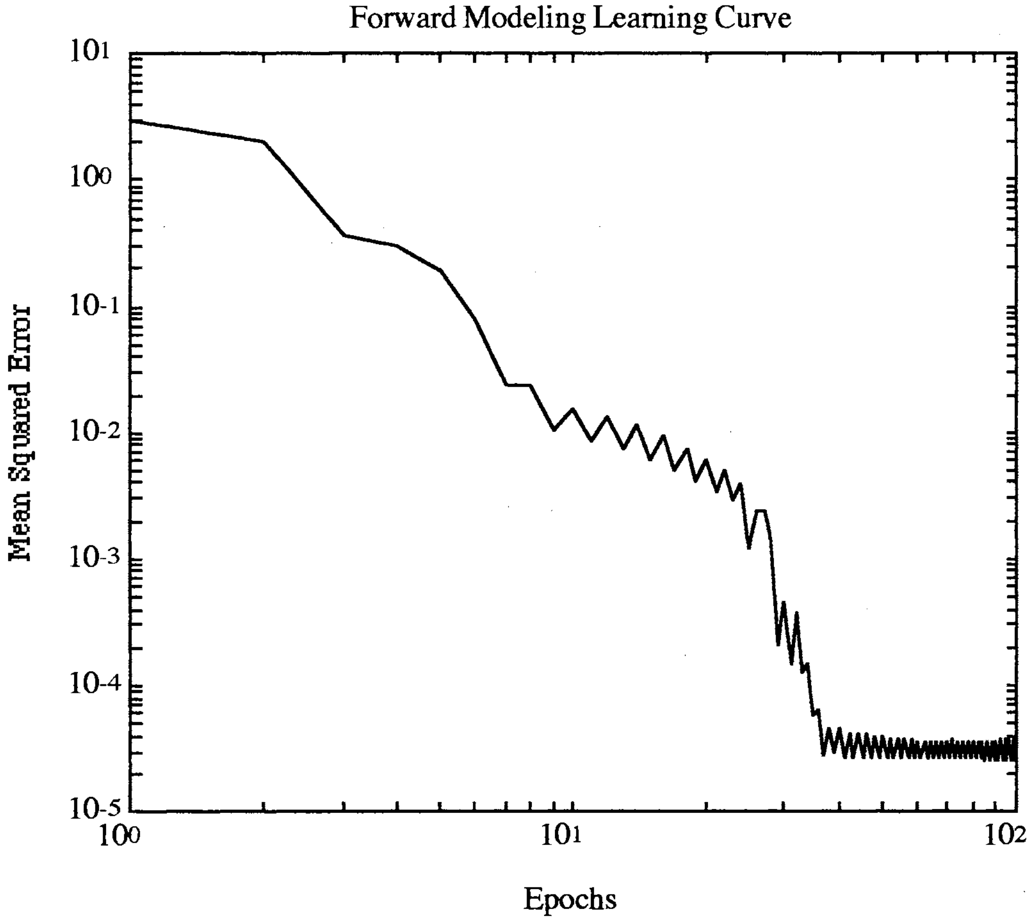


Figure 8.10 The Learning Curve of the Network NN_p

First, we performed the series-parallel tests. The pendulum is initially set to a straight down position ($\theta = 0^o$), then constant voltages of 3.5 and 3.0 volts are applied to the motor to provide two pendulum moving-up motions. The final (steady state) pendulum positions, as shown in Figure 8.11, were about 115^o and 60^o respectively. Two more pendulum moving-down motions from the straight up initial position ($\theta = 180^o$), as shown in Figure 8.12, were performed. The constant voltages applied to the motor were -2.0 and -1.5 volts. One downward motion stopped at -20^o and the other one diverged. In both figures, the real pendulum position (dashed line) is compared to the plant model output (solid line). We observed in both figures, as long as the pendulum motion is within the training range (from 0^0 to 180^o), the network model acted perfectly as the real plant. As shown in the second graph of Figure 8.12, it is surprising that the network model is so accurate, even when the pendulum swings far away from the training range.

Then the more difficult parallel tests were performed. We used the same set-up and procedure as for the series-parallel tests. The results of the parallel tests are shown in Figures 8.13 and 8.14. When compared with Figures 8.11 and 8.12, the parallel test results are certainly not as accurate as the series-parallel test results. Particularly in the parallel tests, the steady state performance is much less accurate than the tracking performance of the network model. We will discuss how the results of the series-parallel and parallel tests of the network model may affect the training and the real-time control of MRAC using dynamic learning in the next section.

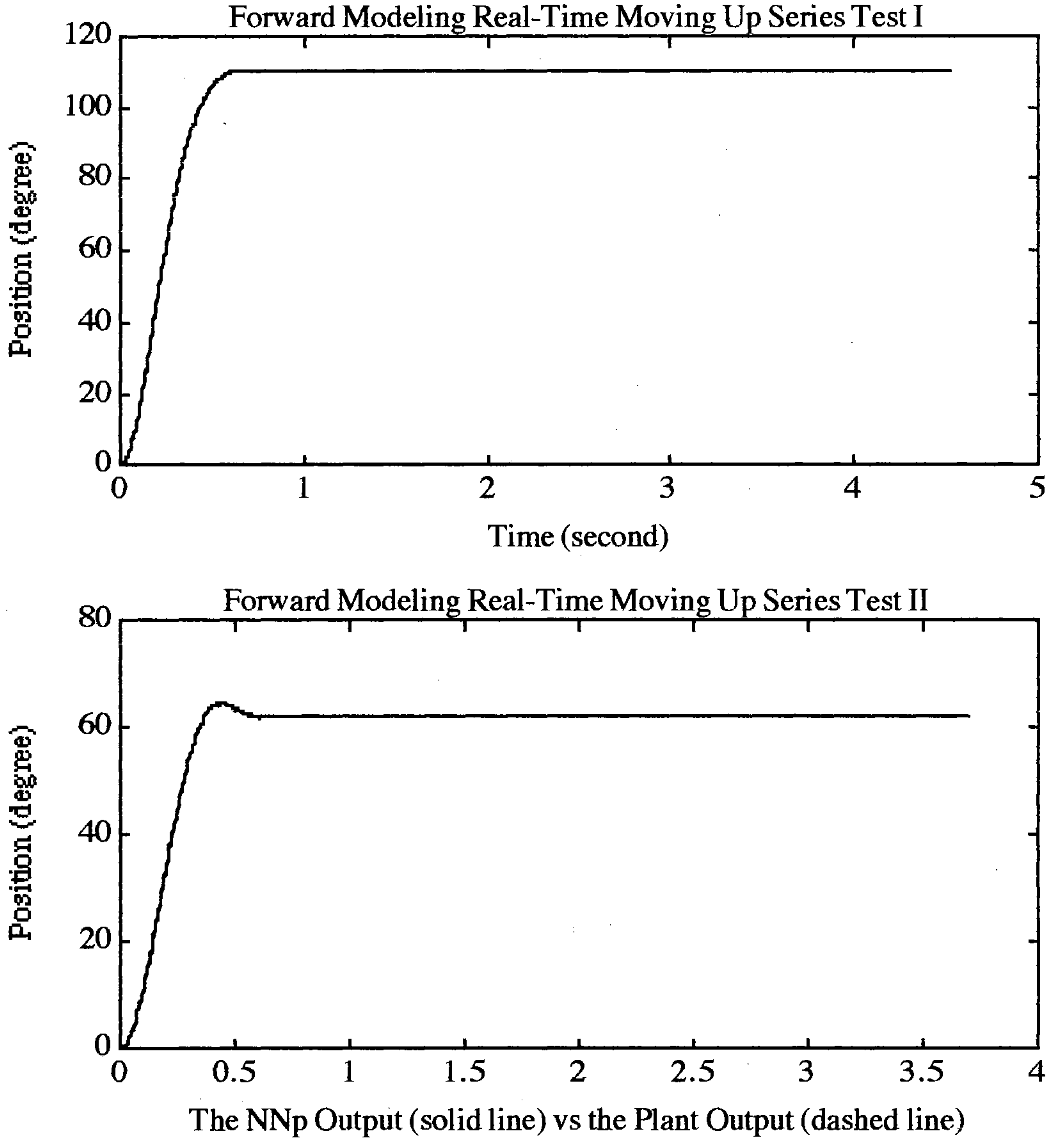


Figure 8.11 The Network Model Moving-Up Series-Parallel Tests

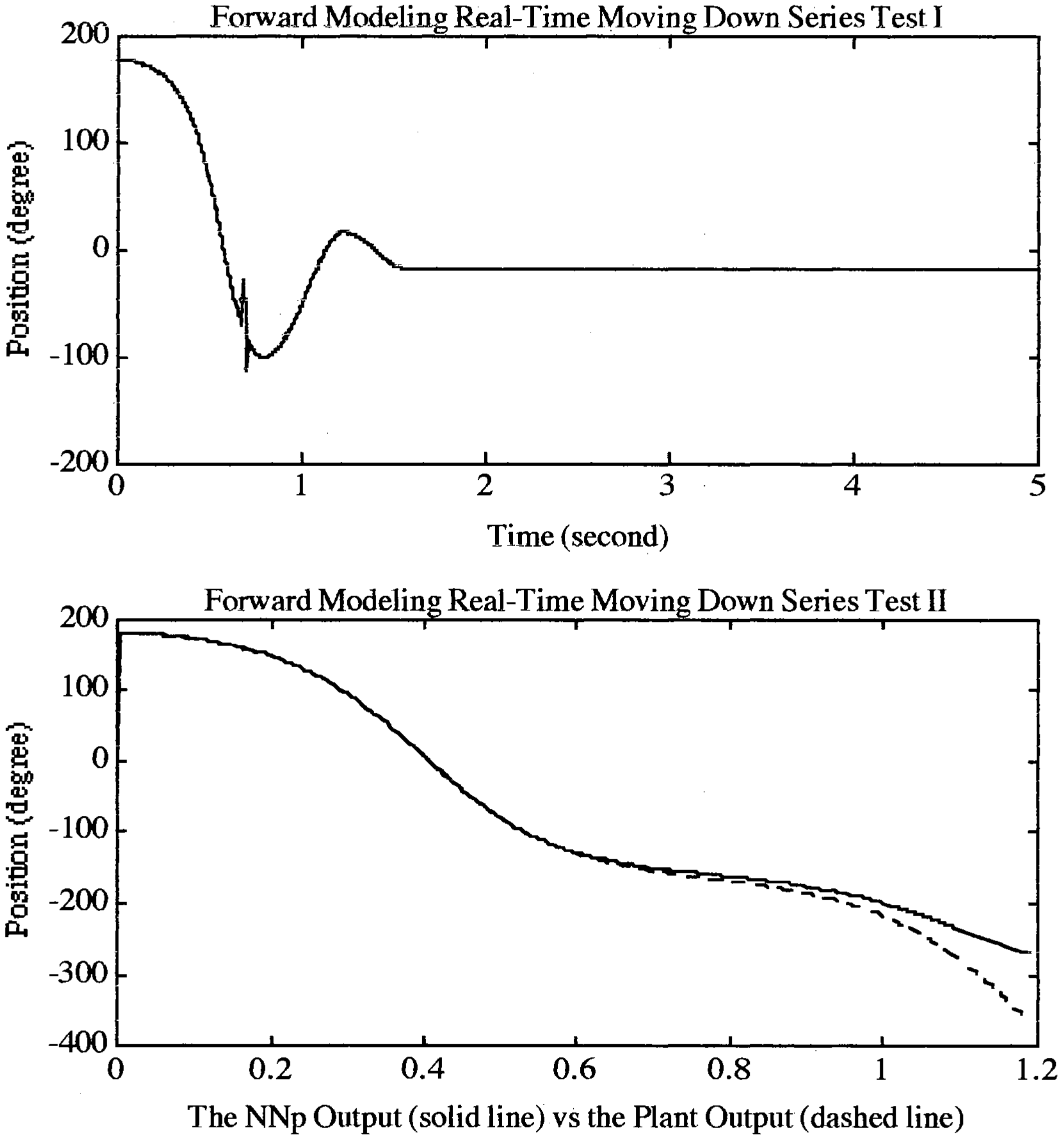


Figure 8.12 The Network Model Moving-Down Series-Parallel Tests

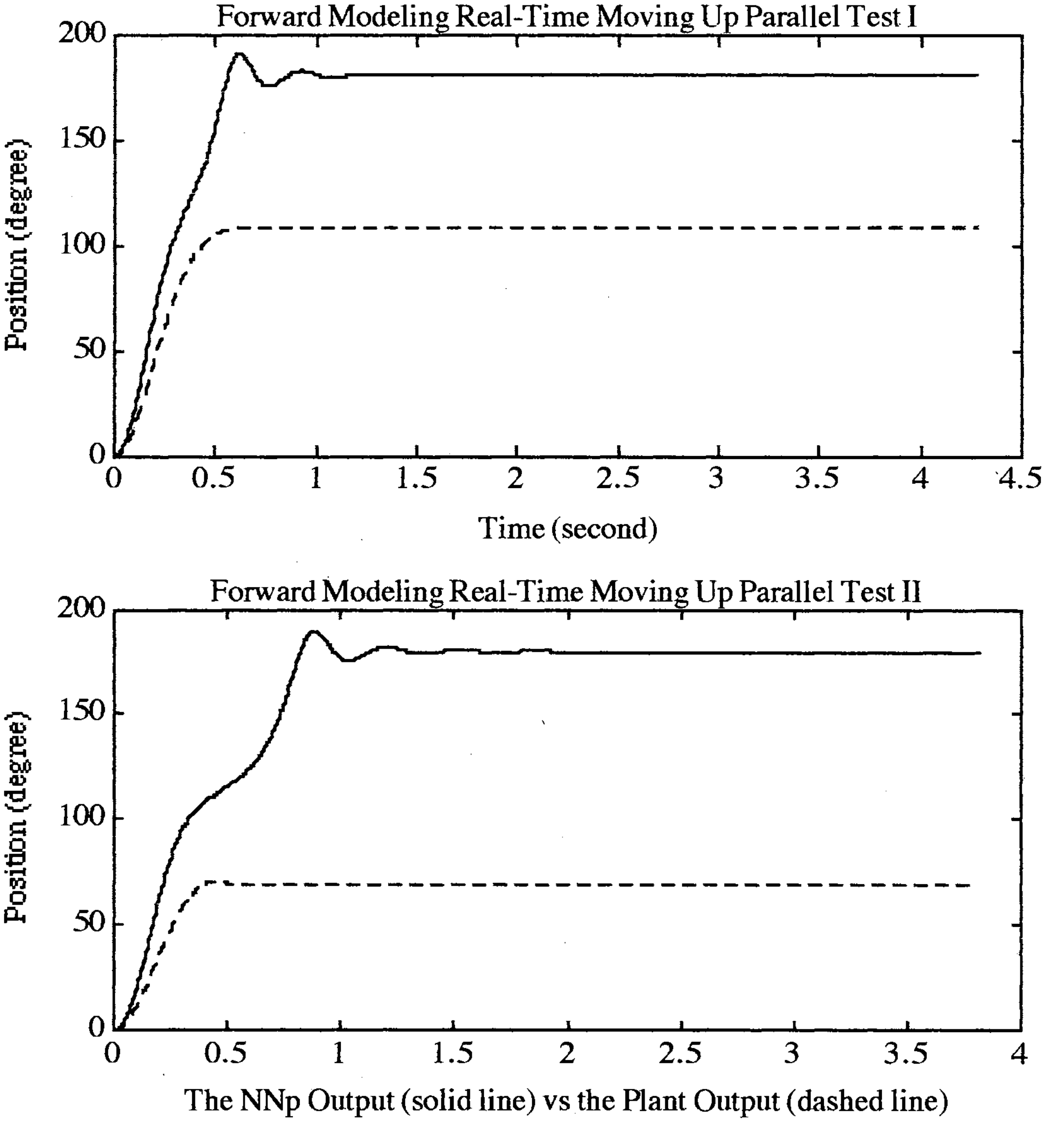


Figure 8.13 The Network Model Moving-Up Parallel Tests

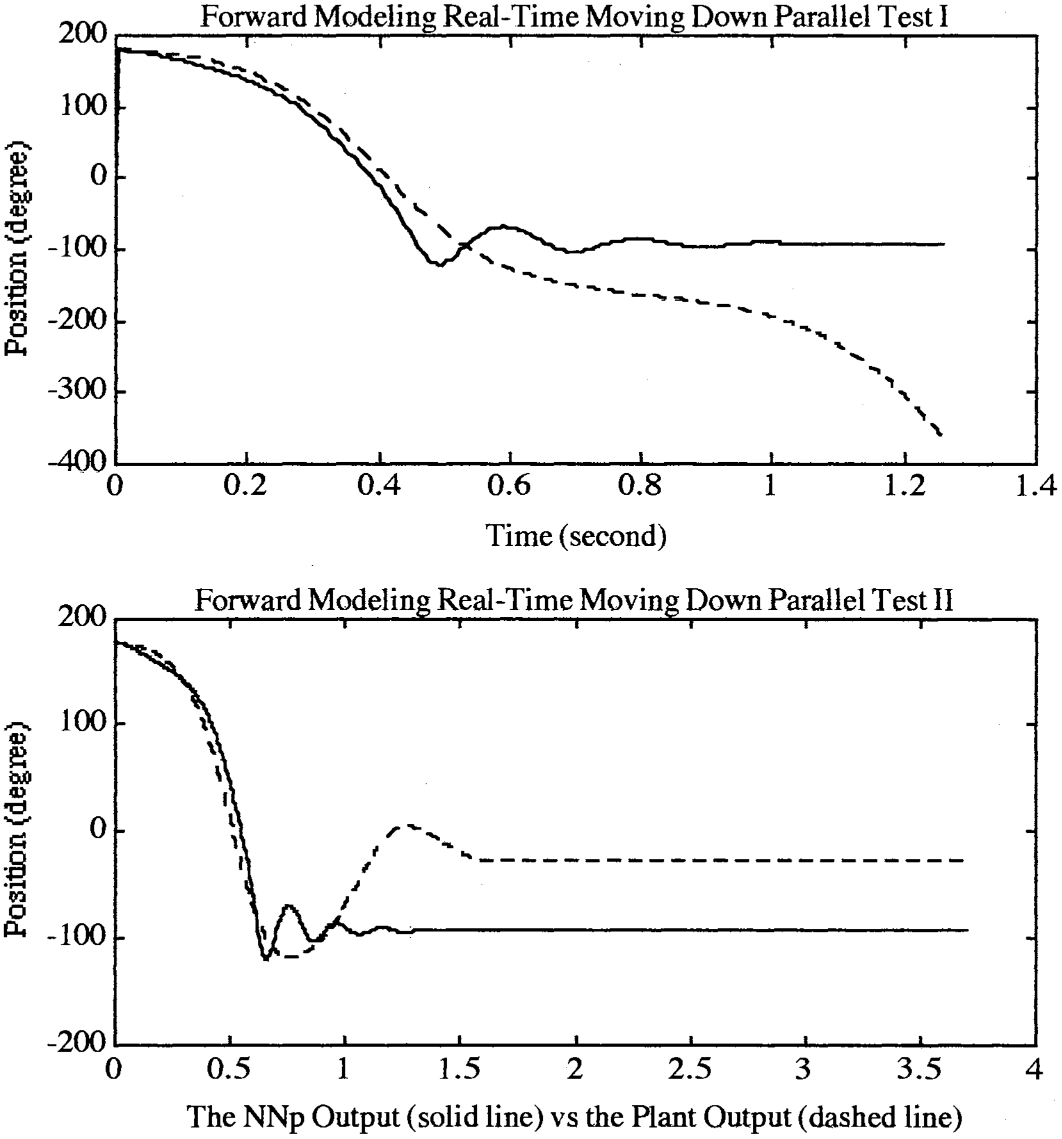


Figure 8.14 The Network Model Moving-Down Parallel Tests

Real-Time Control using Dynamic Learning

We start with a review of MRAC which is described in Chapter 7. Then, the procedure for training the neural network controller for the real physical system is presented. The trained results include the learning curve and off-line tests of the trained network. Finally, the results from real-time control using the trained neural network controller in a real physical system are described.

MRAC and Plant Model

In the original indirect MRAC, as shown in Figure 8.15, the plant supplies the actual plant output to the plant model and the plant model serves as a means to compute the derivatives of the objection function w.r.t. the trained network parameters. However, constrained by the real-time operating environment, we will train the controller off-line. In this situation, a modified indirect MRAC, as shown in Figure 8.16, is used to train the neural network controller for a real physical system.

In order to perform off-line training, the plant model also has to play the role of the real plant. If the plant model only functions as a means for computing the derivative, the series-parallel test, described in the last section, is an accurate enough evaluation method. But in order to act as a real plant, the plant model has to pass the parallel test completely. That is why we commented at the end of the last section that the unsatisfactory outcome of the parallel test of the plant model may affect the training in this section. It suggests that the plant model trained in the last section is not accurate enough to replace the real plant in MRAC dynamic learning. We trained several more plant models, started with different initial network parameters, but the results of parallel tests of these plant models were about the same as the initial one we had. We have no alternative, but to go ahead to train the MRAC controller with the plant model from the last section.

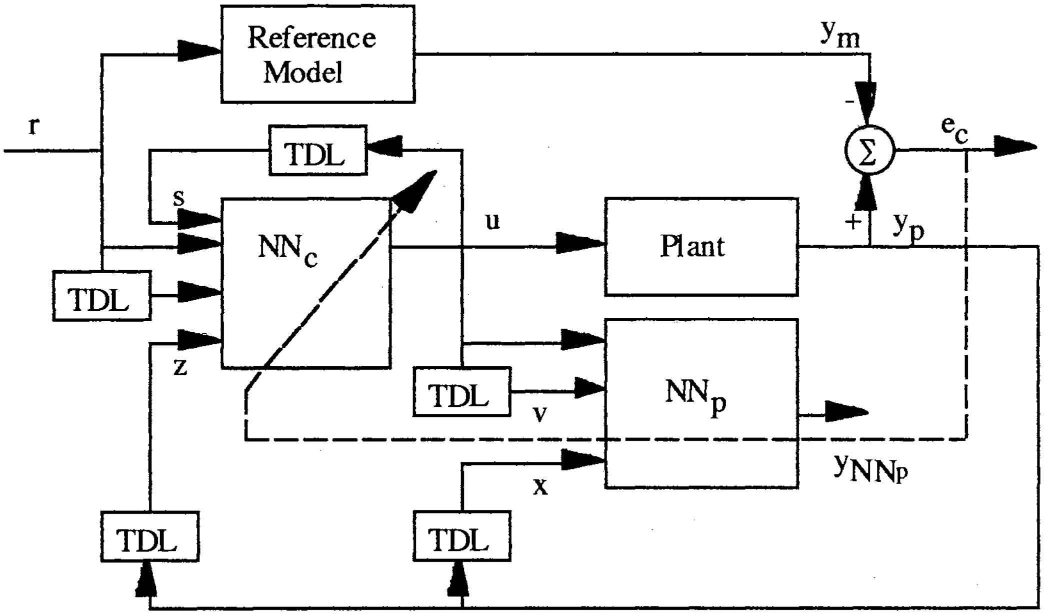


Figure 8.15 Indirect MRAC using Neural Network

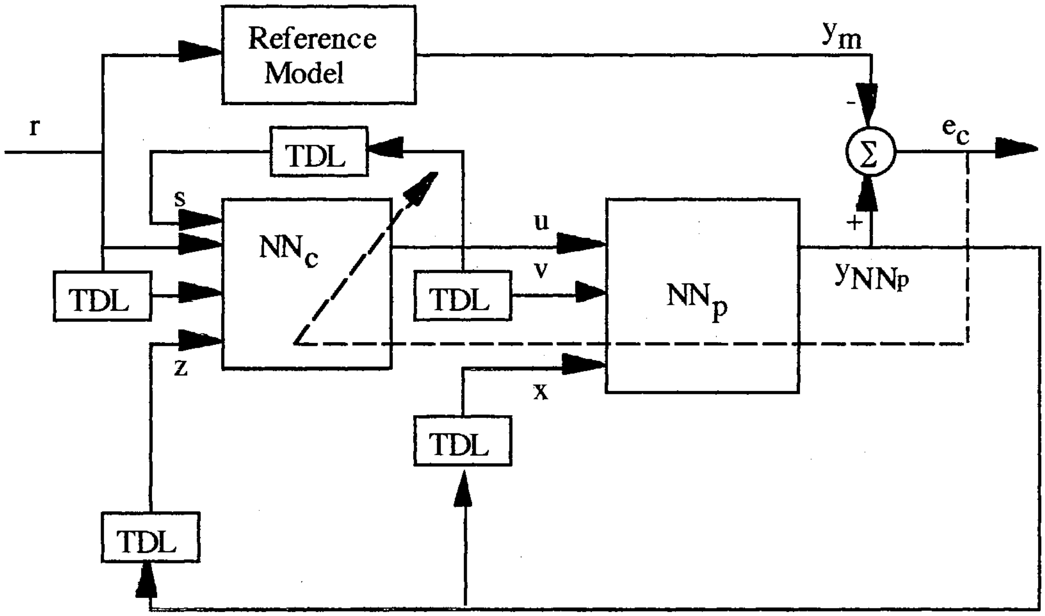


Figure 8.16 Modified Indirect MRAC using Neural Network

Procedure for Training and Real-Time Control

(1) Select a reference model and a neural network for the controller in MRAC. The plant model is from the last section.

(2) Use the same reference input r as in the computer simulation in Chapter 7. Train the network controller using dynamic learning and the TCT method.

(3) Evaluate the trained controller off-line first. If a satisfactory result is obtained, then we will implement the controller in real-time control program.

(4). To evaluate the performance of the off-line trained controller in real-time, we will execute several pendulum up and down motions. We then compare the real plant output to the desired output from the reference model.

The Training and Real-Time Control Results

In the last section, from the parallel tests of the pendulum model, we observed that the pendulum model is almost identical to the real pendulum system in the moving-down tests. Contrary to this, the pendulum model did not act the same as the real pendulum system in the moving-up tests. As we explained previously, in order to perform off-line MRAC training, the plant model has to act exactly like the real plant. Therefore, we came to the conclusion that the controller should be trained by only the moving-down trajectories.

We trained a network pendulum model with 0.05 seconds sampling time. Then, we selected a 5-13-1_92 network to be trained as the controller NN_c in MRAC. The reference model is the same as described in Equation 8.2. After training, we implemented the trained controller in the real-time control program and performed three moving-down tests as shown in Figures 8.17 to 8.19. We observed that the tracking performance of the MRAC neurocontroller is not as good as the regulating performance for all three tests.

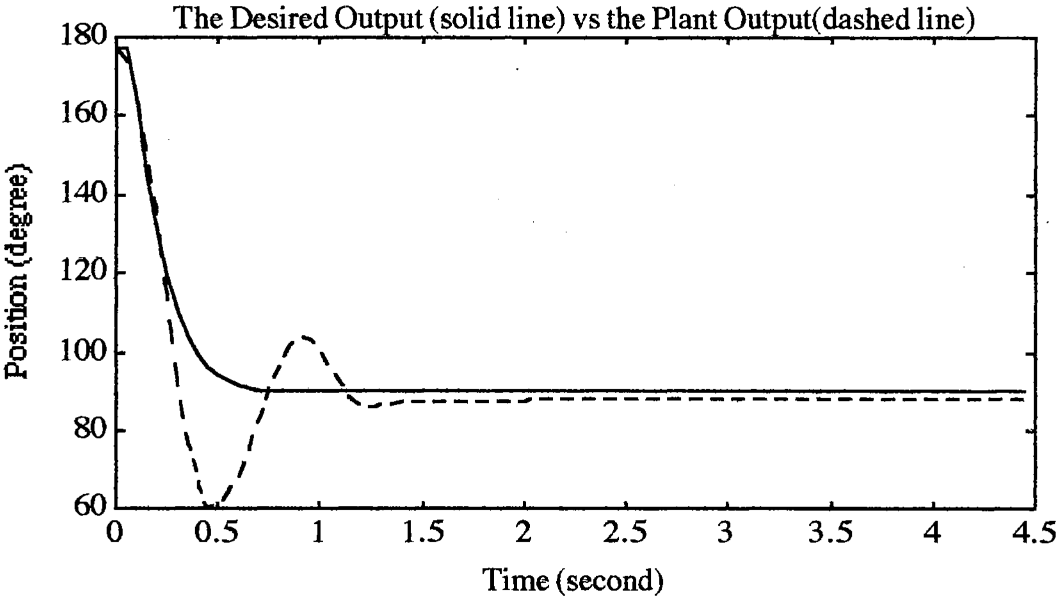


Figure 8.17 The Moving-Down Test I for MRAC Controller

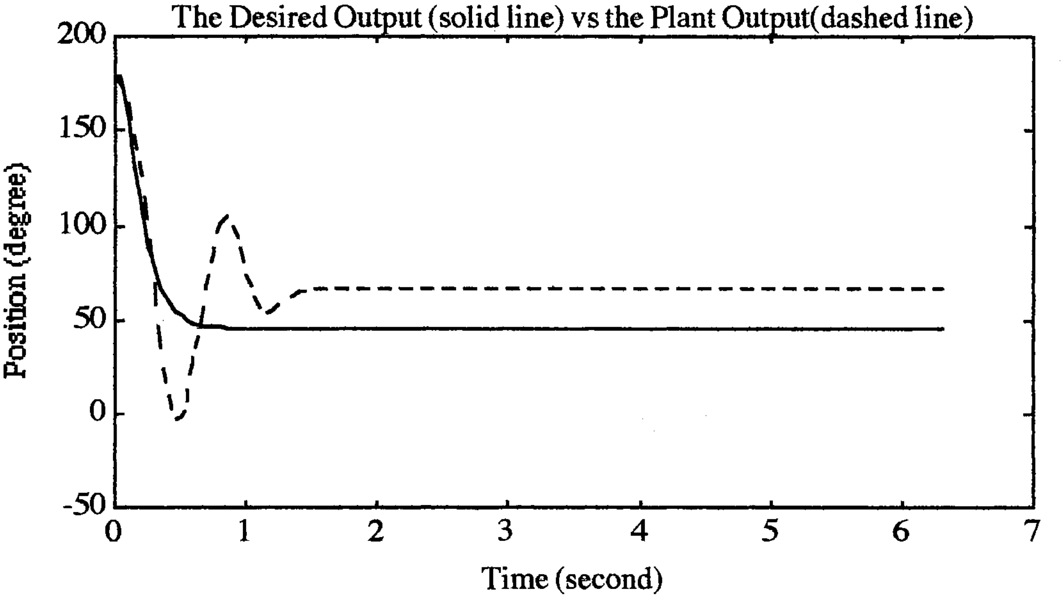


Figure 8.18 The Moving-Down Test II for MRAC Controller

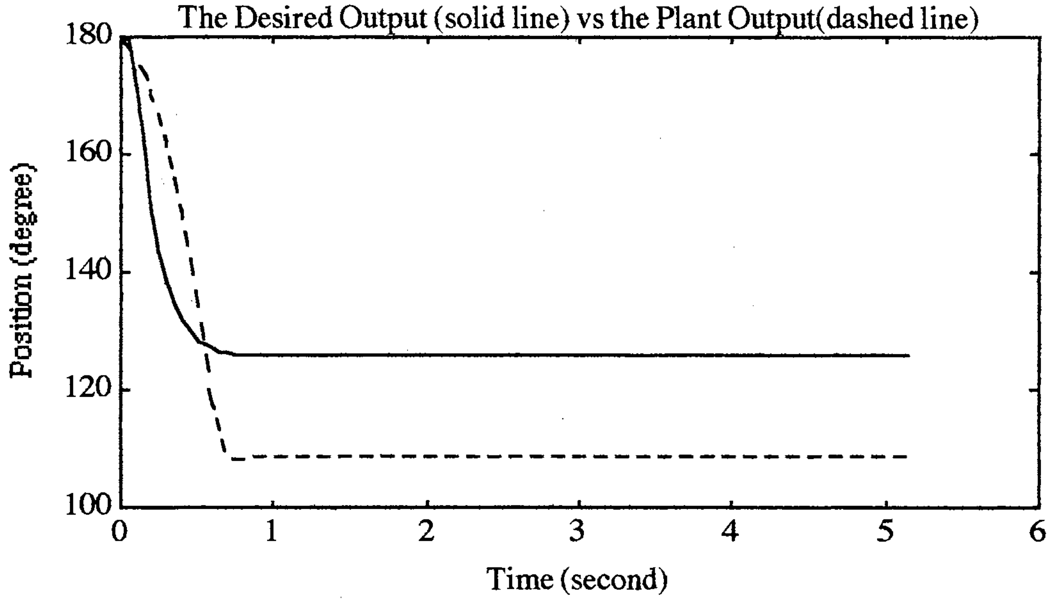


Figure 8.19 The Moving-Down Test III for MRAC Controller

There are offsets when comparing the controlled plant trajectories to the desired trajectories from the reference model, but in general, the off-line trained MRAC controller performs well.

Since the neurocontroller was trained by only the moving-down trajectories, it seems that we should not expect the controller to accurately perform any moving-up movements. It is amazing that, after three moving-up tests (as shown in Figures 8.20 to 22) were performed , we observed that the controller performed as well as in the moving-down tests. This unexpected finding suggests that the neural networks may have been able to generalize from the training data. It is a good subject to be studied in the future.

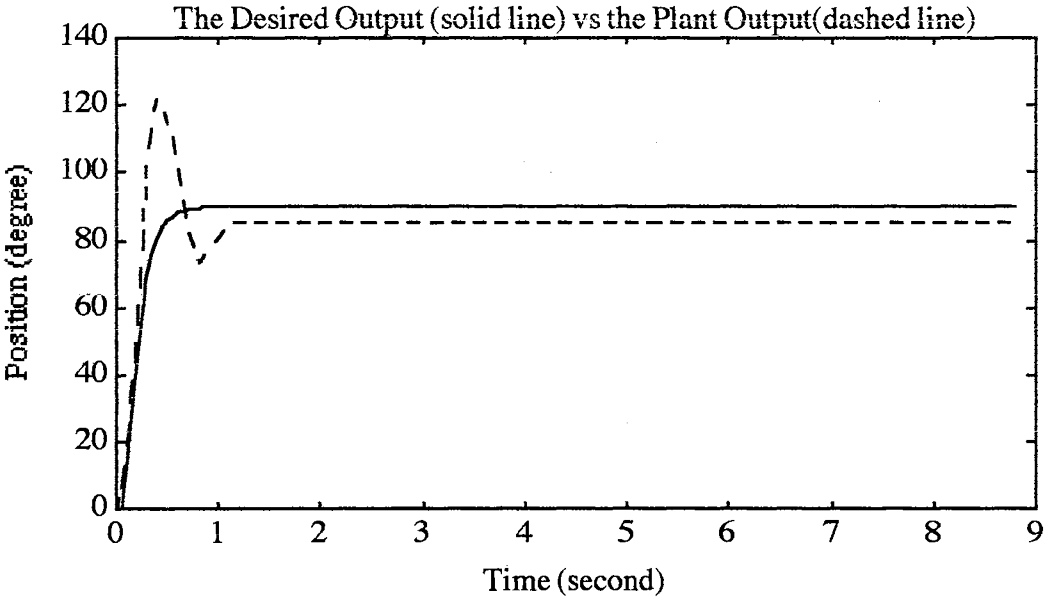


Figure 8.20 The Moving-Up Test I for MRAC Controller

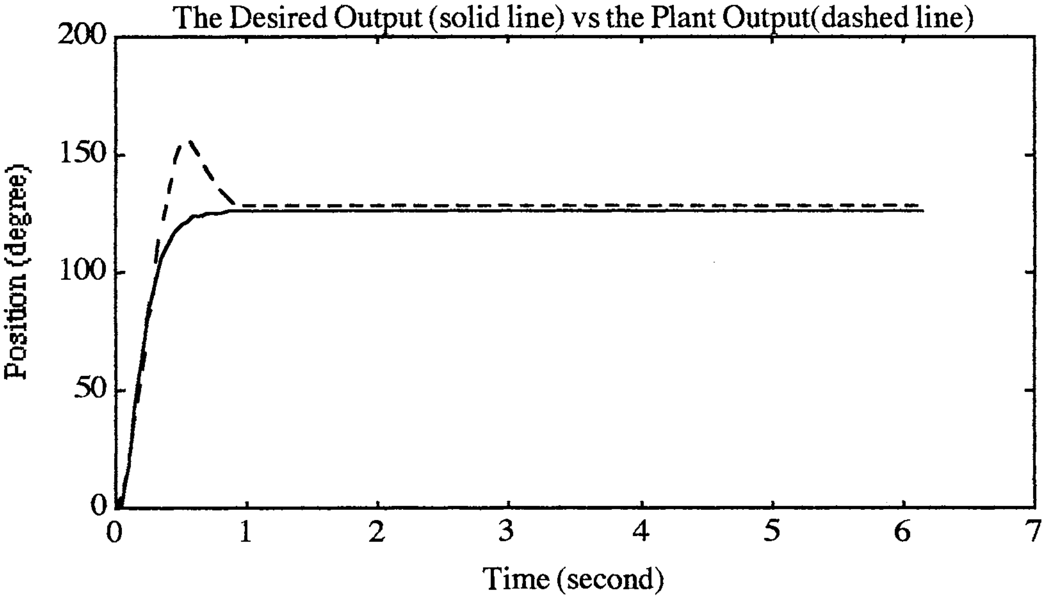


Figure 8.21 The Moving-Up Test II for MRAC Controller

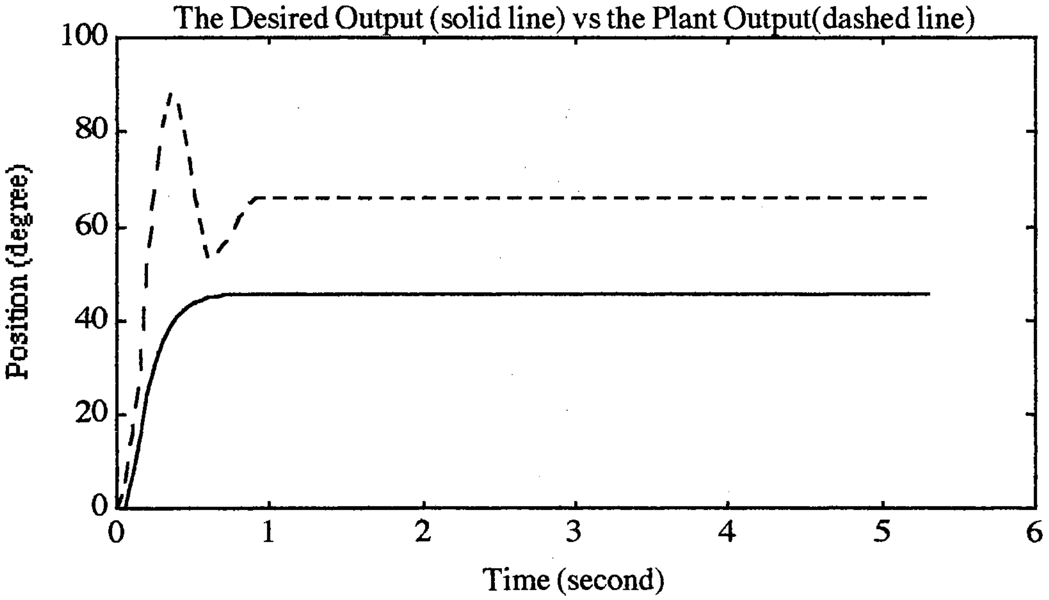


Figure 8.22 The Moving-Up Test III for MRAC Controller

CHAPTER IX

SECONDARY TRAINING METHODS

In Chapters 4, 5 and 6, we described the underlying algorithms for the static and dynamic learning methods. They are the Marquardt optimization technique using basic backpropagation and the Marquardt optimization technique using FP/BTT algorithms. We call these the primary training methods. Unfortunately, based from our empirical experience, these primary training methods did not always guarantee a satisfactory training result. Therefore, so-called "secondary" training methods were proposed to aid the primary methods in training the neural networks.

In this chapter, two secondary training methods -- the ATS method for static learning and the TCT/TLI method for dynamic learning -- will be presented. Computer simulations will be performed for the ATS method. Using the same initial conditions, the simulations will compare training results using the ATS method to training results using a common general approach. Since there is no known general approach in dynamic learning, only the description of the TCT/TLI method is given in this chapter. Computer simulations and a real-time application using the TCT/TLI method can be reviewed in Chapters 7 and 8 respectively.

The ATS Method

Two training data sets are required for the alternating training data set (ATS) method. They can be obtained by the same data collection process. However, each has a different set of initial conditions. If only one data set is available then it can be broken into two data sets for the training. We then alternate the presentation of these two data sets to the network at each learning epoch. The adjusted parameters of the network from one data set in one learning epoch, therefore, become the starting parameters for the other data set at the next learning epoch.

The ATS Method vs the General Approach

The ATS method is first compared to a general approach in static learning. Then, we will test whether the ATS method can prevent the problem of data overfitting or overtraining. Data overfitting or overtraining is the tendency of trained networks to learn specific details in the training data instead of the general properties of the underlying function.

To avoid the overtraining problem, the commonly used general approach also requires two data sets. In the general approach single set is used for training and the other set for evaluating the trained network weights after each learning epoch. The evaluation set monitors the learning process, which only utilizes the training data set. We will find from our later simulations that there are two typical patterns for the evaluation curve (the plot of the mean squared error versus epoch number for the evaluation data set) when the learning curve is falling. The evaluation curve either levers off or arrives at a minimum point begins to increase. This minimum point is where overtraining starts and the learning process should be stopped.

Computer Simulation I

We will use the swinging pendulum system identification problem described in Chapter 5 for the computer simulations of the ATS method and the general approach. A 4-5-1_31 neural network is selected and trained as the plant identifier. We will first try to establish a general pattern for the learning curves exhibited by the ATS method and the general approach. Then, in order to compare the performance of the two approaches, the parallel test method is used to evaluate the trained results.

In order to observe any general patterns, three different initial conditions (in terms of the initial network weight set) are given. The resulting learning curves after 1000 epochs for each of the three cases are shown in Figures 9.1 through 9.3.

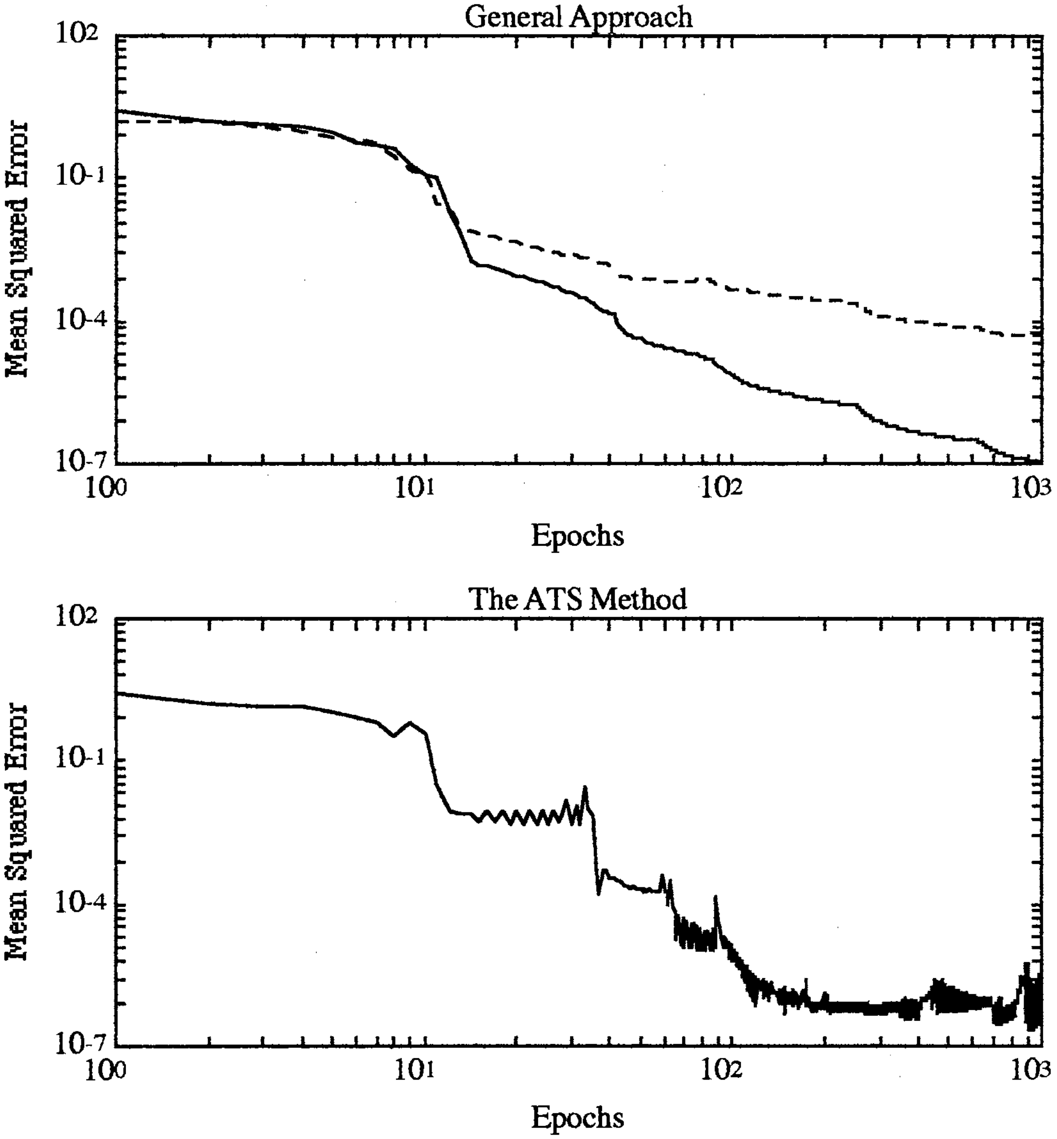


Figure 9.1 The Learning Curves for the General Approach vs. the ATS Method for Case 1

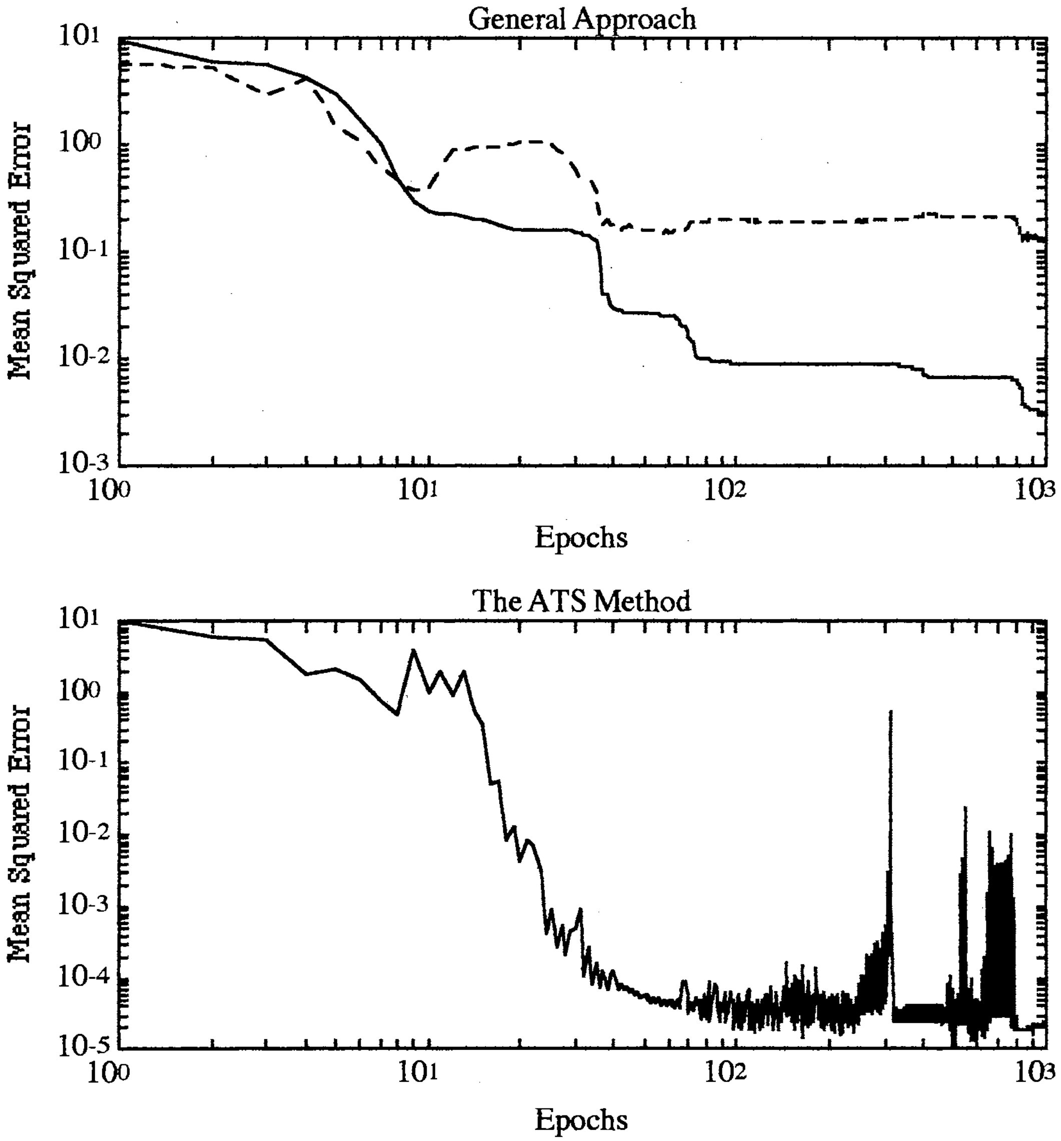


Figure 9.2 The Learning Curves for the General Approach vs. the ATS Method for Case 2

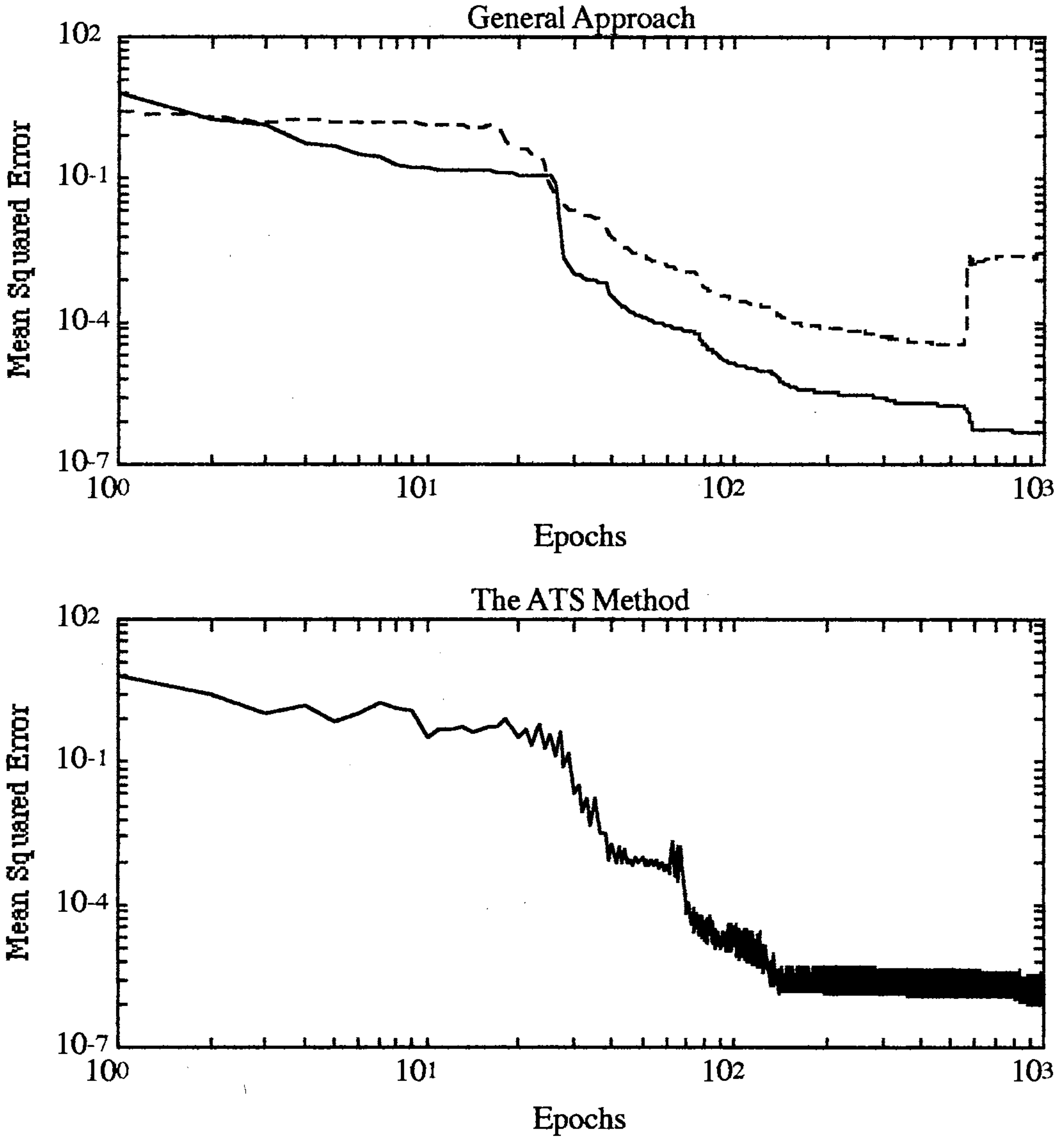


Figure 9.3 The Learning Curves for the General Approach vs. the ATS Method for Case 3

From these three figures, we observed that the ATS learning curve may have some unusual "spikes" (particularly in Case 2). However, in each of the cases the learning curve is still going down at end of training. For the general approach, the evaluation curve (dashed line) is pulling away from the learning curve (solid line) in first two cases and in Case 3 the evaluation curve reaches a minimum point and then starts to climb. These observations indicate that the problem of overtraining exists in the general approach. To solidify these observations, we need to test all of the trained results from both the approaches by the higher standard parallel test method.

The results of the parallel tests performed on the trained results from both the ATS method and the general approach are shown in Figures 9.4 through 9.6. The test results follow the order of presentation of the cases in Figures 9.1 to 9.3 respectively. For each of the three cases, two initial conditions (in terms of position and constant motor voltage) are given and tested. The solid line stands for the output of the reference model (the desired plant output). The dashed and dotted lines represent the outputs of the neural network models of the pendulum trained by the ATS method and the general approach respectively. The solid line and the dashed line are almost indistinguishable in all of the tests. This means that the ATS method has good trained results for all three cases. In contrast to the general approach, the dotted and solid lines are definitely distinguishable in the first test of Case 1. In addition, the tests performed in Case 2 are complete failures. From the above simulations and tests, we can conclude that the ATS method not only never fails, but it also is always more accurate than the general approach.

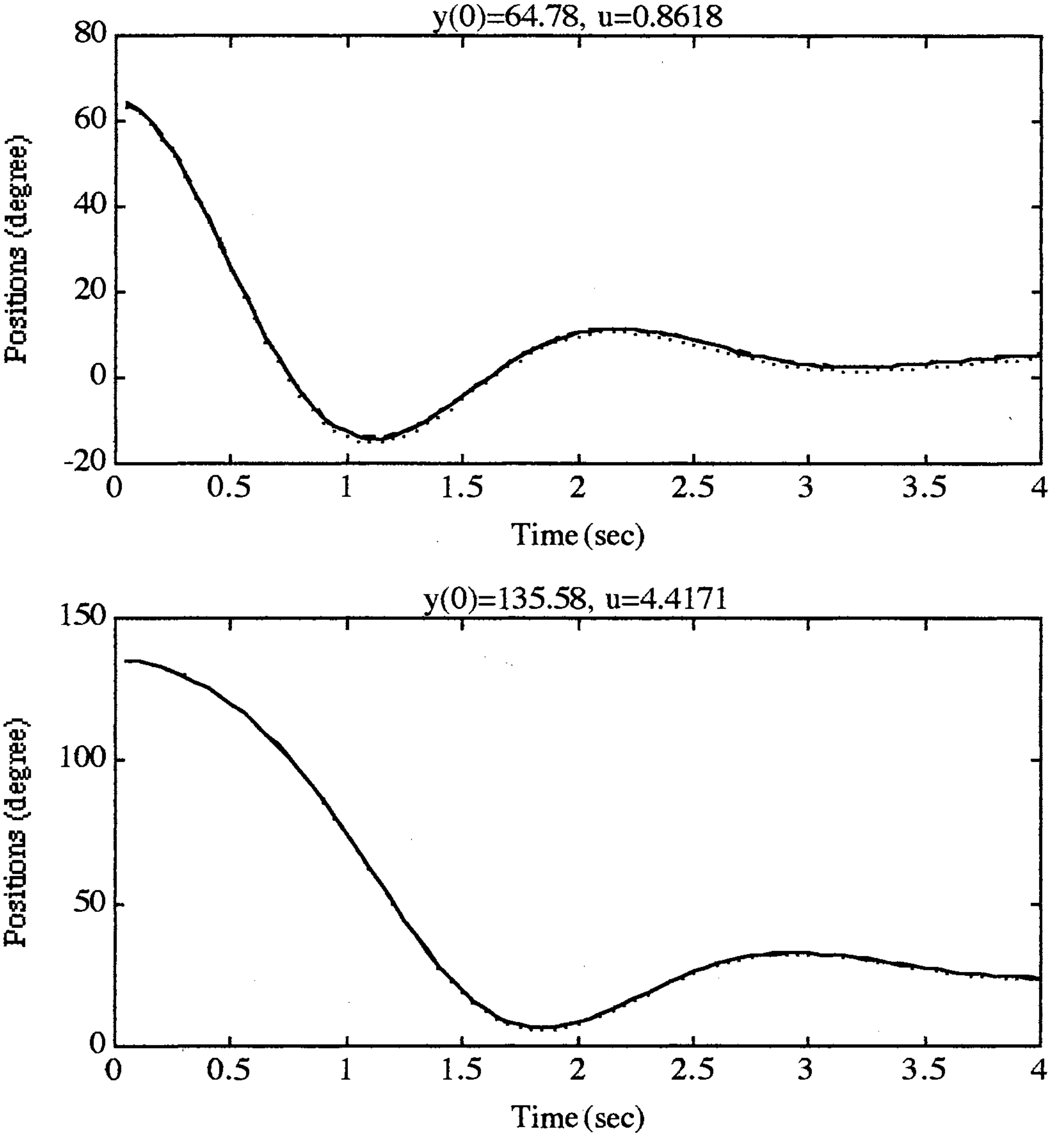


Figure 9.4 Results of the Parallel Tests for Both Approaches For Case 1

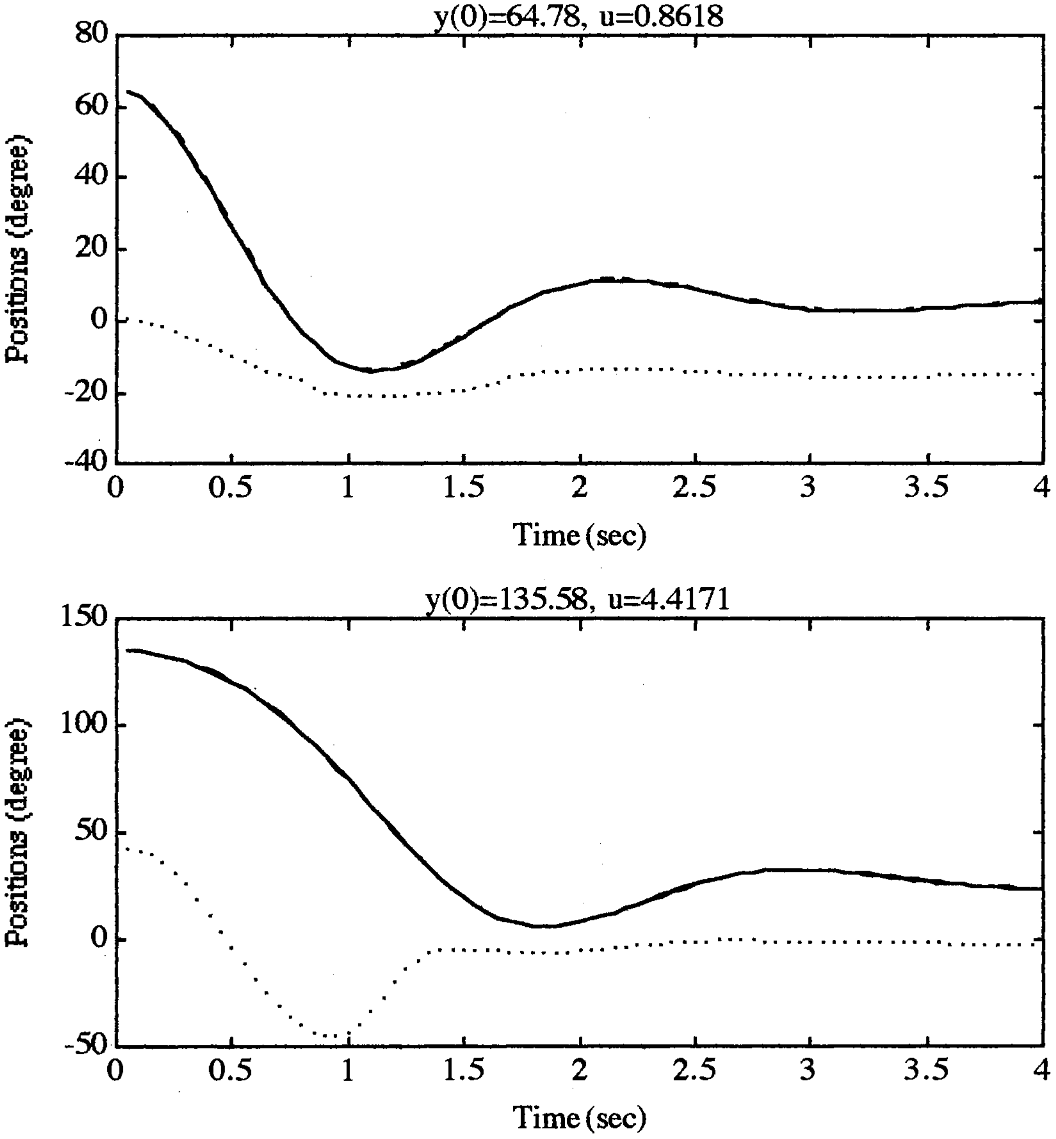


Figure 9.5 Results of the Parallel Tests for Both Approaches For Case 2

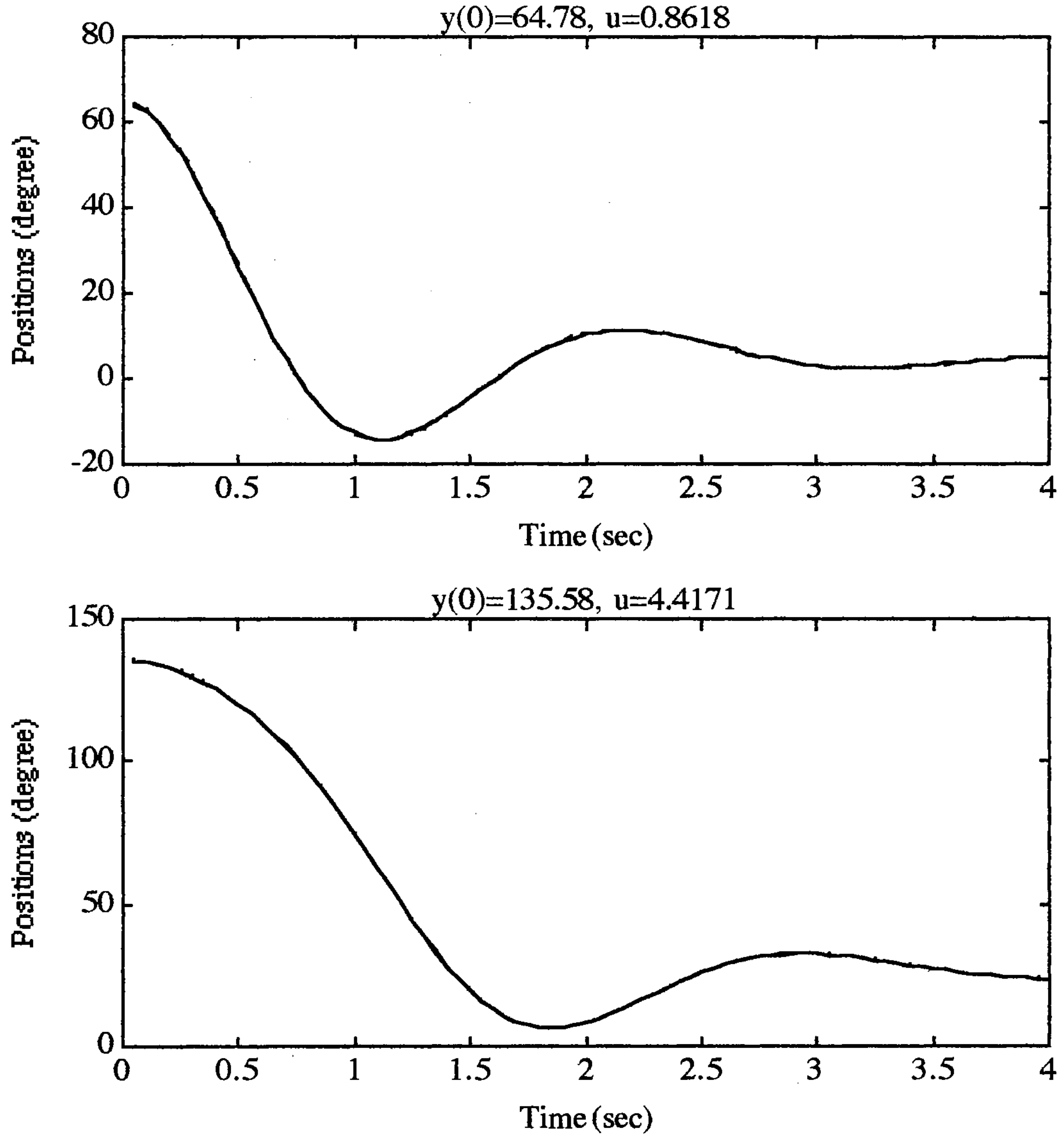


Figure 9.6 Results of the Parallel Tests for Both Approaches For Case 3

The ATS Method vs Single Set Training

Since all of the available training data are utilized in the ATS method, it is not fair to compare it to the general approach in which only part (the training data set) of the available data are used. So we will now use all of the available data as one complete data set to train the networks and compare these results to the ATS method. It is obvious that using a single complete data set training is not practical because we then have no evaluation data set to detect the presence of overtraining. However, we want to examine whether the ATS method not only can prevent overtraining, but also can obtain a training outcome as good as the outcome obtained from single set training. To do this, we will compare the ATS method to the impractical single set training method. In order to detect overtraining in single set training, a "third" data set is generated for use in evaluating the training results.

Computer Simulation II

To compare the ATS method to the single set training method, the same computer simulation set up as in Simulation I is used. We will first try to establish the general pattern of the learning curves exhibited by these two different approaches. Trainings for three different initial network weight sets were performed for both methods. The results for the three cases are shown in Figures 9.7 to 9.9 respectively. In these figures, the solid and dashed lines are stand for the learning and evaluation curves respectively.

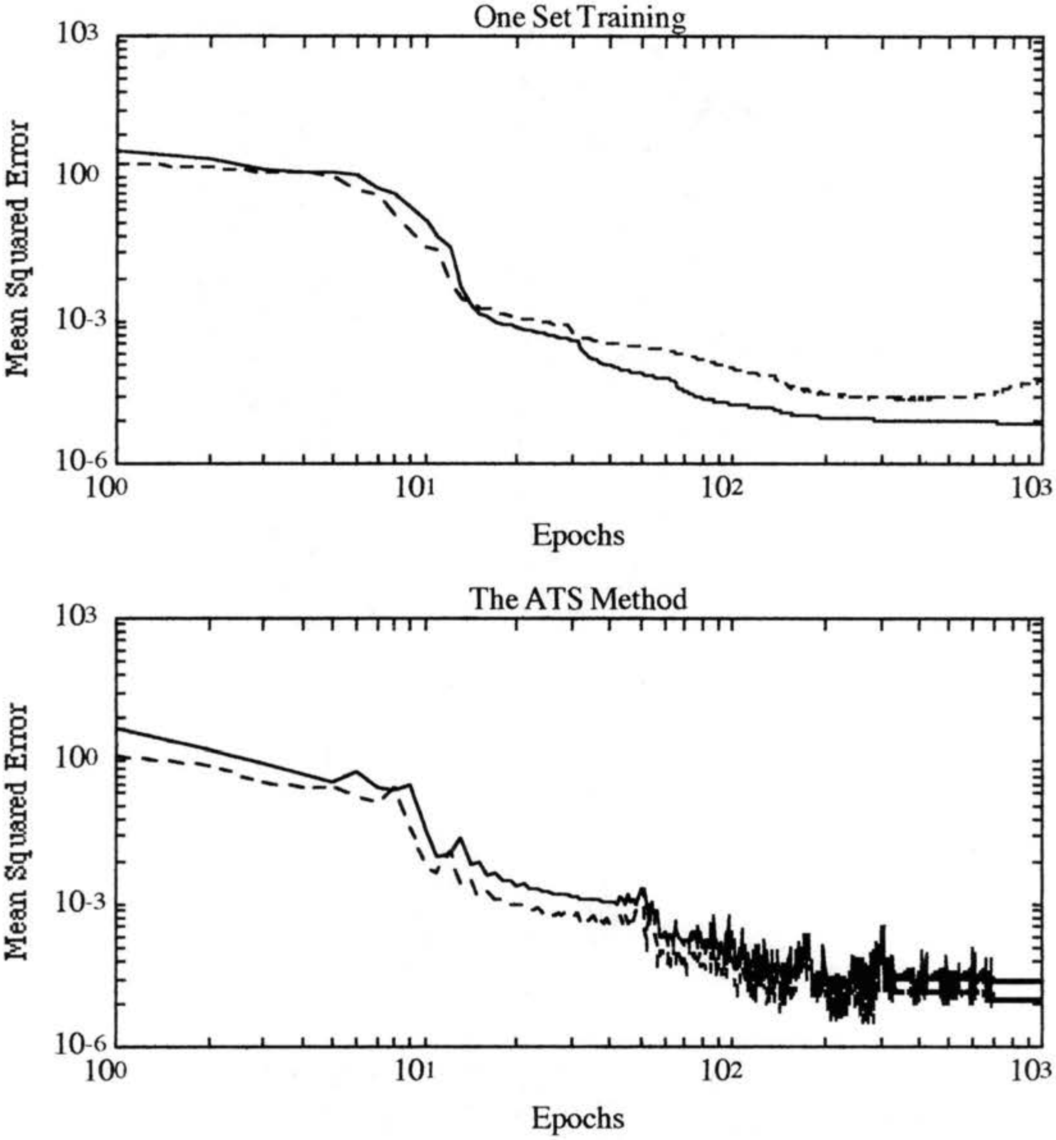


Figure 9.7 The Learning Curves for the Single set Training Method vs. the ATS Method for Case 1

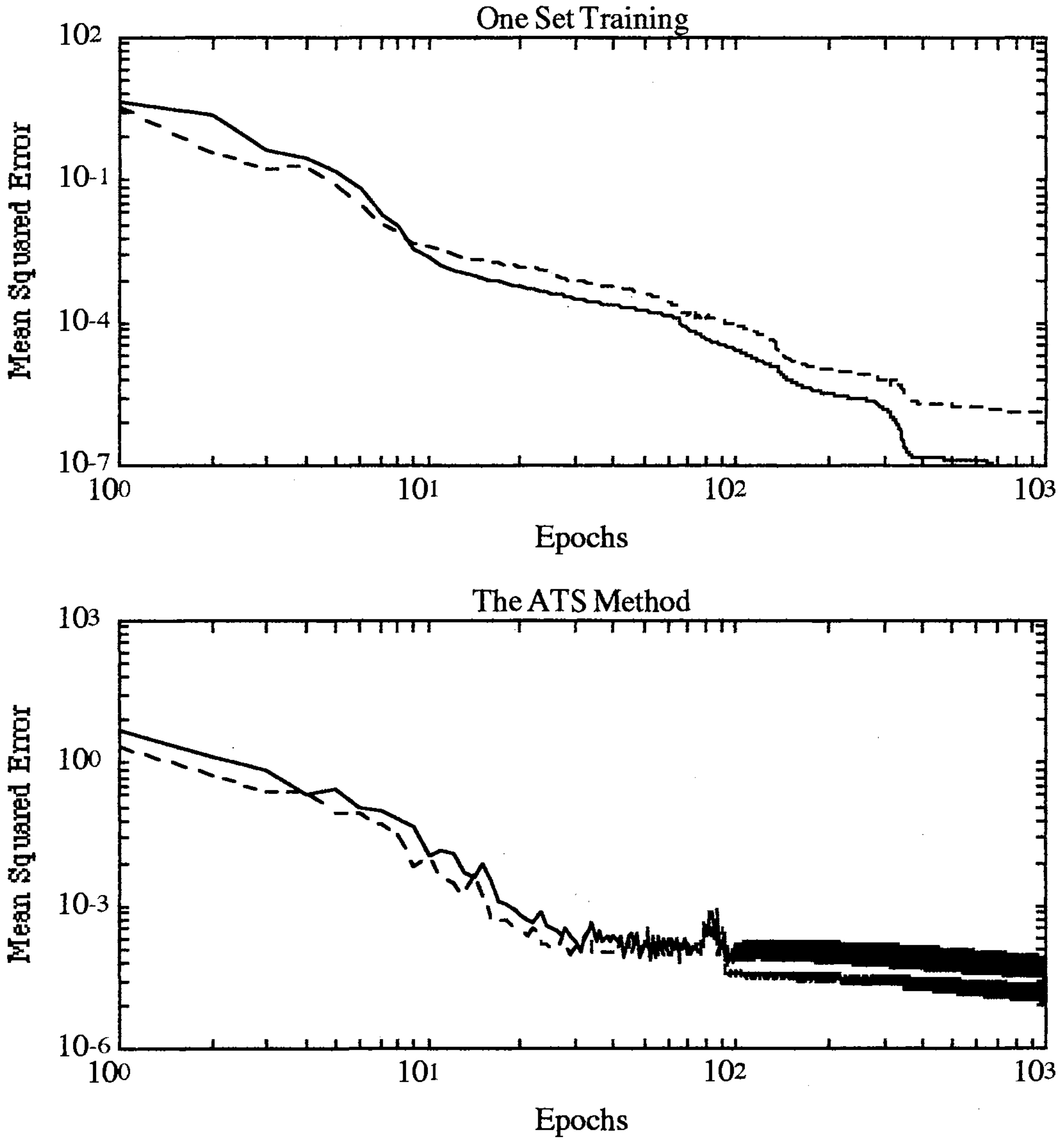


Figure 9.8 The Learning Curves for the Single set Training Method vs. the ATS Method for Case 2

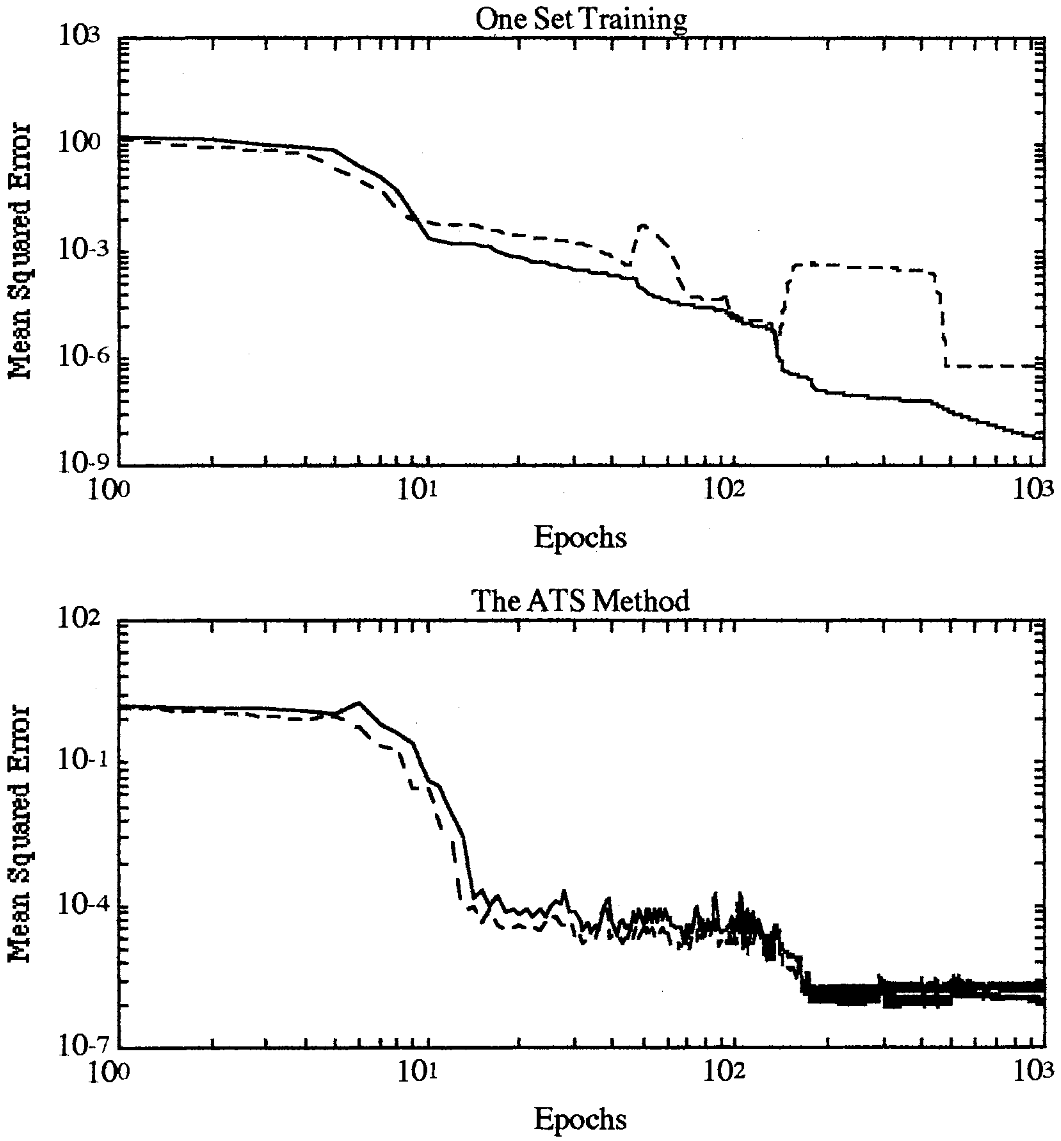


Figure 9.9 The Learning Curves for the Single set Training Method vs. the ATS Method for Case 3

Next, the parallel tests were performed for both approaches in each case. The test results all look alike for all three cases with one example for Case 1 shown in Figure 9.10. In that figure, the desired output (solid line), the ATS output (dashed line) and the single set training output (dotted line) all become one indistinguishable line. This means that both the ATS method and single set training method generate satisfactory training results at the end of 1000 epochs. This also answers the question that was raised at the beginning of this section -- the ATS method is as accurate as the single set training method, and eliminates overfitting.

Comments

From the simulations for this pendulum identification problem, we conclude that the ATS method has no overtraining problem and uses all available data for training. The training curves of the general approach and the single set training methods both tend to overtrain as the learning process goes forward. This problem is solved by the compliment training between two alternating data sets that occurs in the ATS method. However, further research is needed. For example, such work might try increasing the number of the alternating data sets (more than two) and testing the applicability of this ATS method to other static learning problems.

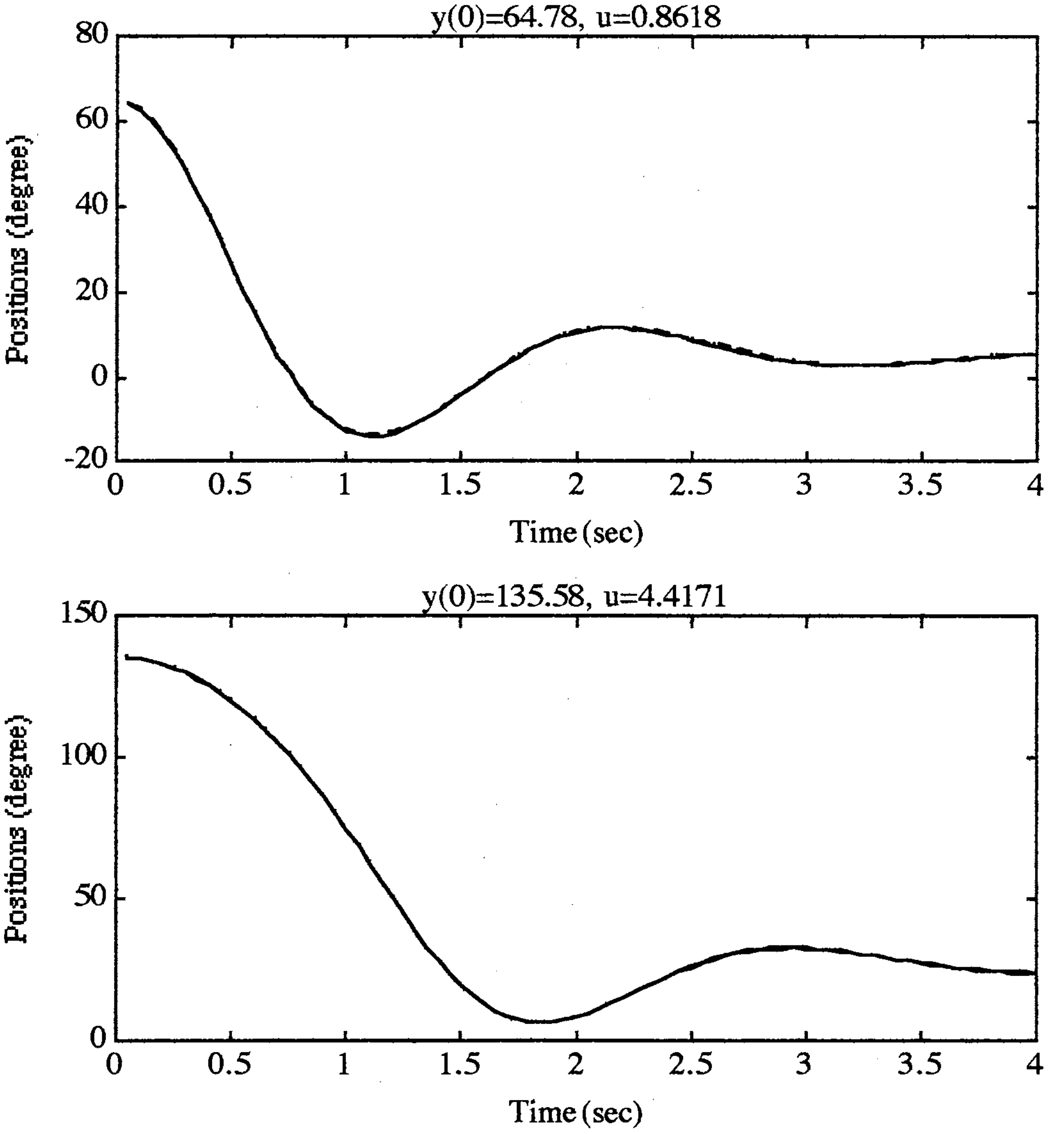


Figure 9.10 General Results of the Parallel Tests for Both Approaches for All Three Cases

The TCT/TLI Method

The TCT (trajectory cross training)/TLI (trajectory length increasing) method is used as a secondary training method in dynamic learning. We will explain the reason that we connect the TCT and the TLI methods as a single method after we first describe the TCT method.

The idea of the TCT method is derived from the ATS method for static learning. The basic idea behind both methods is complimentary training based on two (or more) training data sets. In terms of system trajectory, each training data set contains two characters -- the underlying function and the specific details of the training data set itself. The specific details of each trajectory may vary, but they all possess a common characteristic -- the underlying function. In using the ATS method for static learning, we alternate the training data set one at a time. In the TCT/TLI method, we stack up a number of trajectories to create one big data set for training. By doing so, we expect that the common character, the underlying function, will be more dominate in the learning process than the distinctive character of each trajectory. We expect that the wanted common character, instead of the unwanted distinctive characteristics, of each trajectory will be extracted by the learning process.

One of the most common problems in dynamic learning is the huge error between the desired output and the neurocontrolled plant output at the beginning of the learning process. This is caused by the random selection of the initial weights of the neurocontroller. Thus the output of the controller, which is the control input of the plant, is unpredictable. The range of the control input can be unreasonably high or low from the controller. Therefore the behavior of the controlled plant is unpredictable too. In most of experiments we conducted, the output error was always big at the beginning of the learning process. To deal with this problem, we came up with the TLI method. The idea of the TLI method is to keep the data points (the length of the system trajectory) as low as possible at the beginning stage so that the sum of the squared error is also kept low. At this stage, the network parameters can be adjusted based on the small amount of training data, then the network output is moving down toward a reasonable range. After the initial stage, we can increase the number of data points stage by stage as the network parameters converge toward the solution. So, strictly speaking, the TLI method is more of an idea than a method. Besides, in most cases the TLI method is a necessary approach in dynamic learning. Therefore, we have connected the TCT and the TLI methods into a single method.

We have already demonstrated the implementation of the TCT/TLI method in Simulation 3 of Chapter 7. In the following computer simulations, we will emphasis just the TCT method.

Computer Simulation III

Using the TCT method in dynamic learning, we first designed a number of runs (each run generates a single trajectory) in each learning epoch. The number of the runs is determined by trial-and-error in different applications. Then, we stacked these trajectories up to use as a single set for training. The neural network training problem used in this subsection is the same as Simulation 4 of Chapter 7. In this problem a neural network controller (4-5-1_31) is trained to control a second-order linear plant (a linearized pendulum system) while responding to a given reference model.

In most linear plant applications, no over-range error problem happens at the beginning of training. Therefore, the TLI method is not needed. In this situation, we are only concerned with the selection of data points for one training data set. Two cases have been studied for the effect of varying the number of data points in the training set. The total number of training data points in one learning epoch is 20 and 100, for Cases 1 and 2 respectively. For both cases, we compared the one trajectory training method to the TCT method (two trajectories per learning epoch). The two trajectories in the TCT method are divided from the trajectory used in the one trajectory training method. In Case 1, we trained the controller for 2,000 epochs. The resulting learning curves are shown in Figure 9.11 and an enlargement of Epochs 1,901 to 2,000 is shown in Figure 9.12. From the enlargement picture, we observe that the learning curve for the one trajectory training method has larger amplitude oscillations as compared to the amplitude of the oscillations for the TCT learning curve. This result suggests that the TCT method generates a more accurate trained controller than the one trajectory training method.

To back up this observation, we use the parallel test method to evaluate the trained controllers from both of the training methods.

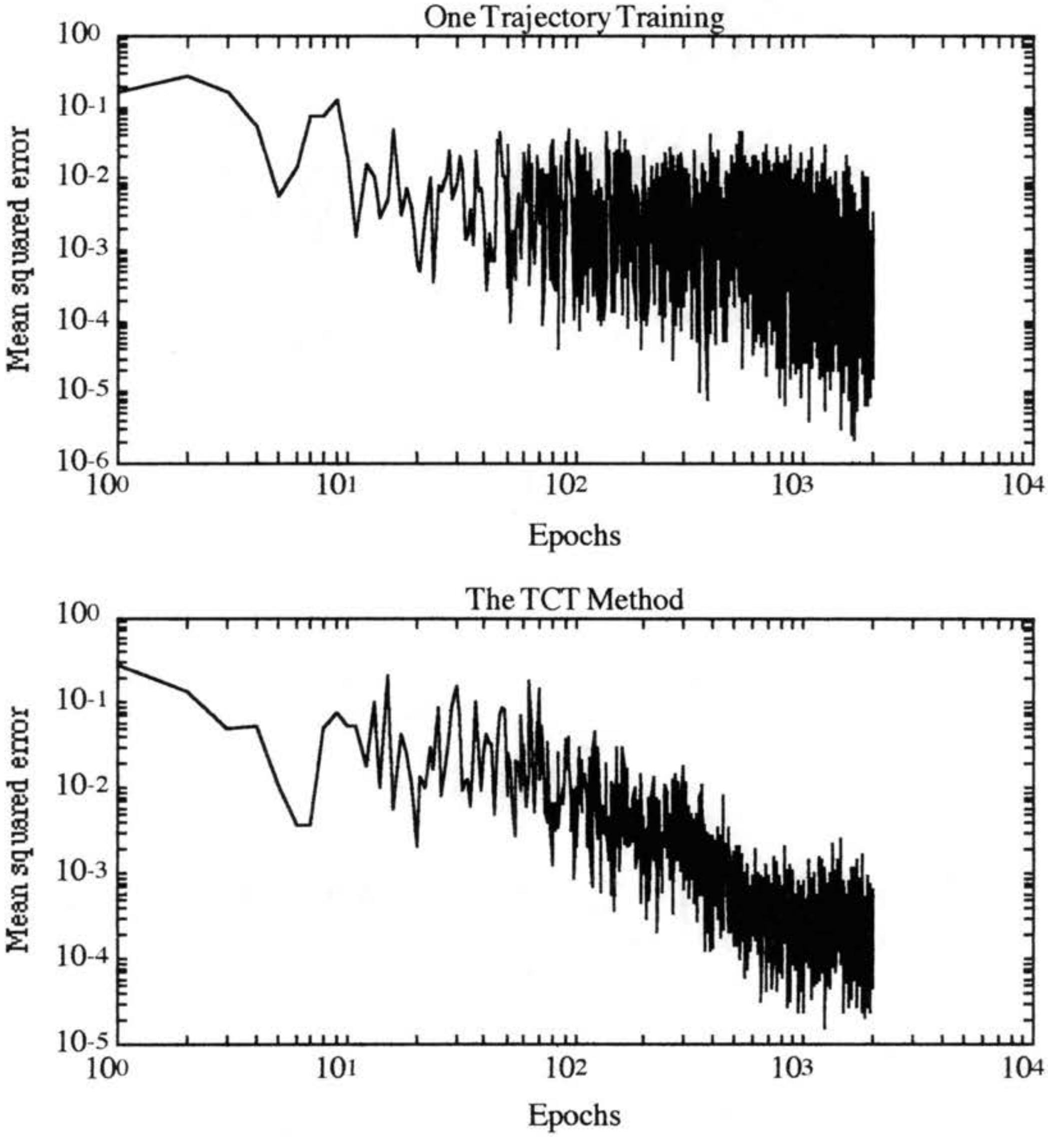


Figure 9.11 Learning Curves of One Trajectory Training vs the TCT Method for Case 1

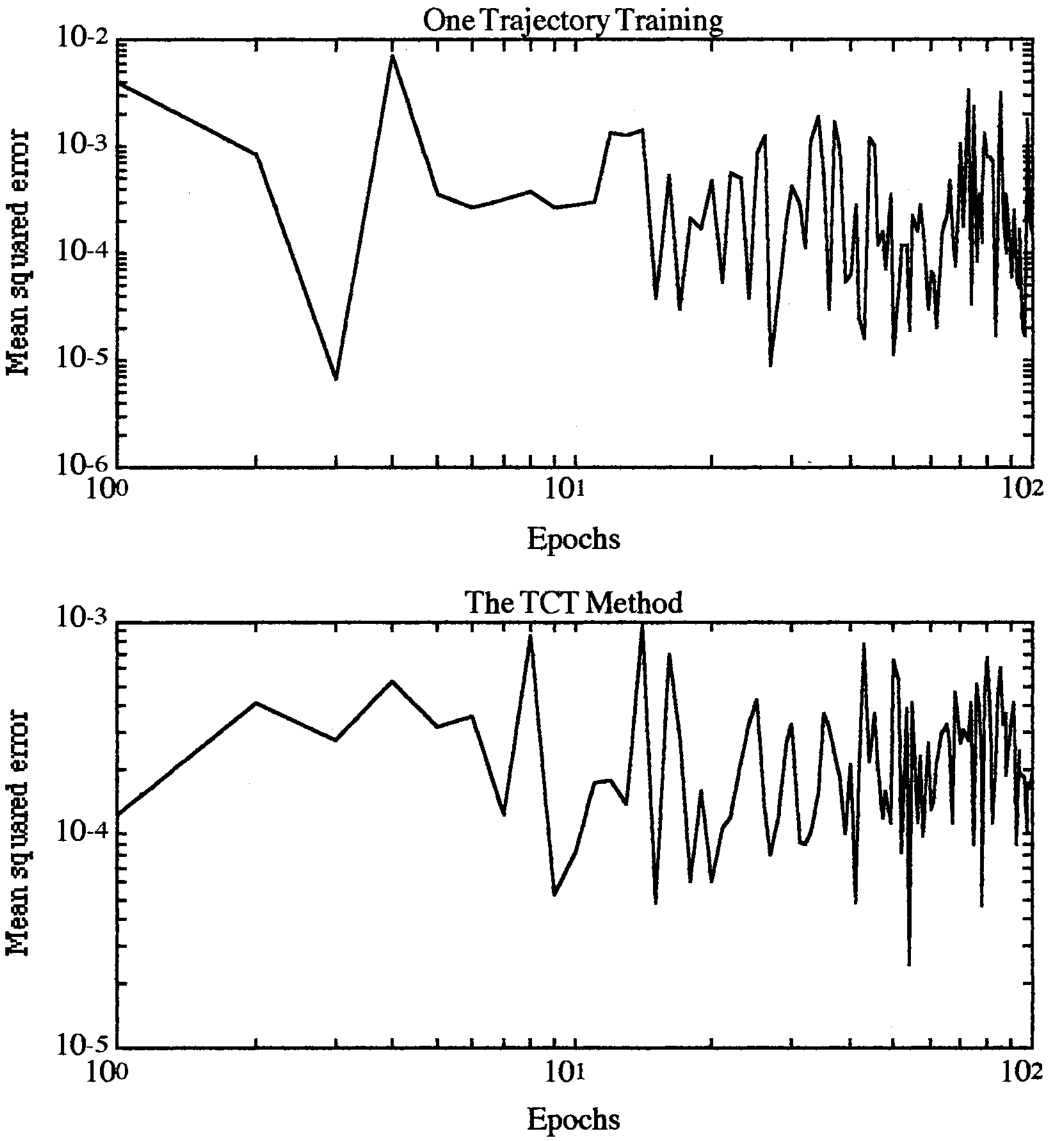


Figure 9.12 Enlargement of the Learning Curves for Epochs 1901 to 2000 from Figure 9.11

The parallel test results are shown in Figure 9.13. In the figures, the solid line stands for the desired output and the dashed line represents the controlled plant output. We found that the test result from the TCT method is more accurate than the test result from the one trajectory training method. This is consistent with our observation on the amplitude of the oscillations from both of the learning curves.

In Case 2, the total number of data points is extended to 100. After 2,000 epochs learning, as shown in Figures 9.14 (the learning curves) and 9.15 (the enlargements), we found that the one trajectory training method has smaller amplitude oscillations than the TCT method. In addition, the learning curve of the one trajectory training method reaches a lower mean squared error than the TCT learning curve at end of 2,000 epochs. Given randomly chosen initial conditions, the parallel tests of the trained controllers from both the training methods were performed and the results are shown in Figure 9.16. The test results confirmed the observations made in the learning curves that the one trajectory training method is more accurate than the TCT method in this case.

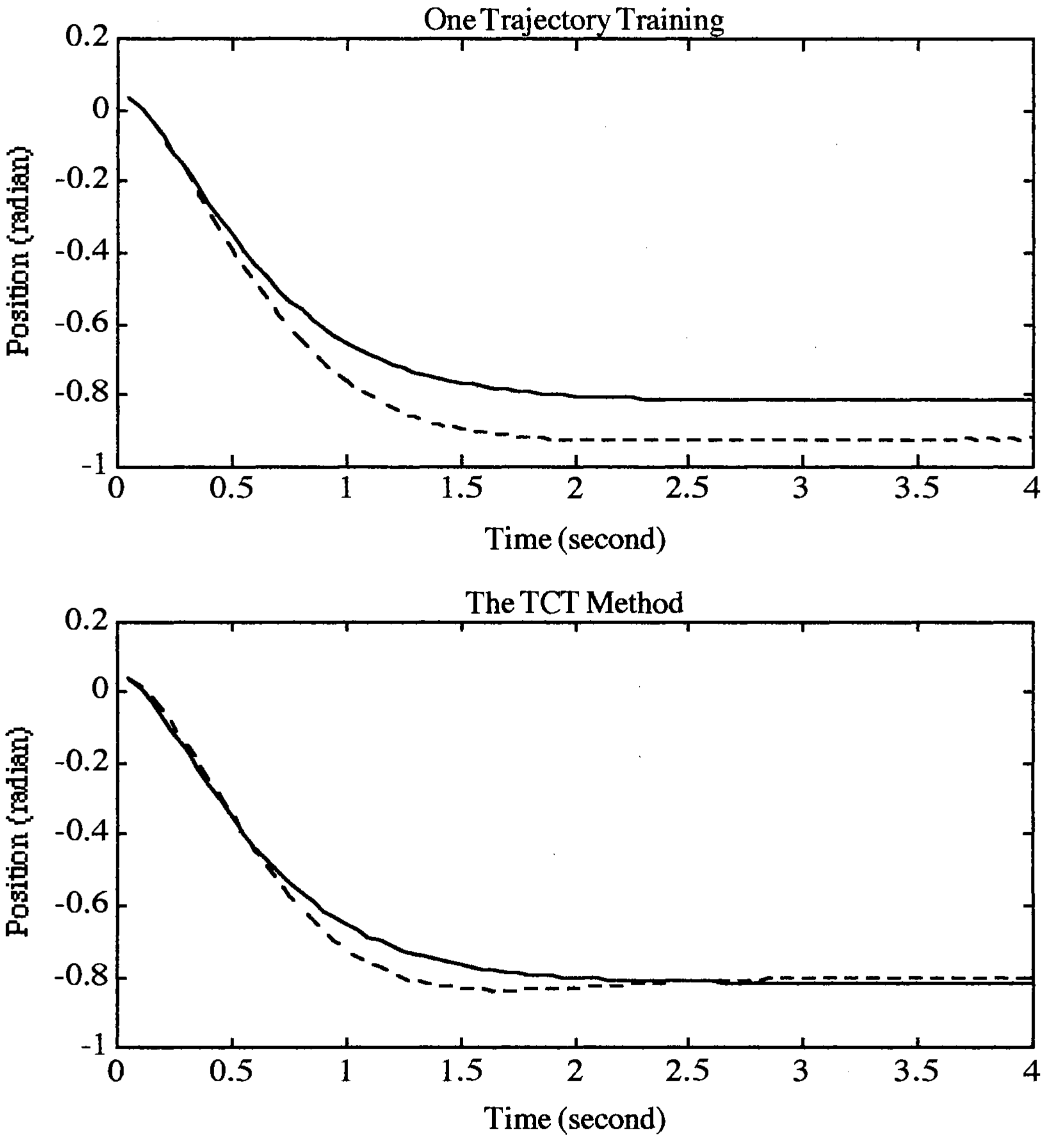


Figure 9.13 The Parallel Test Results for the Trained Controllers for Case 1

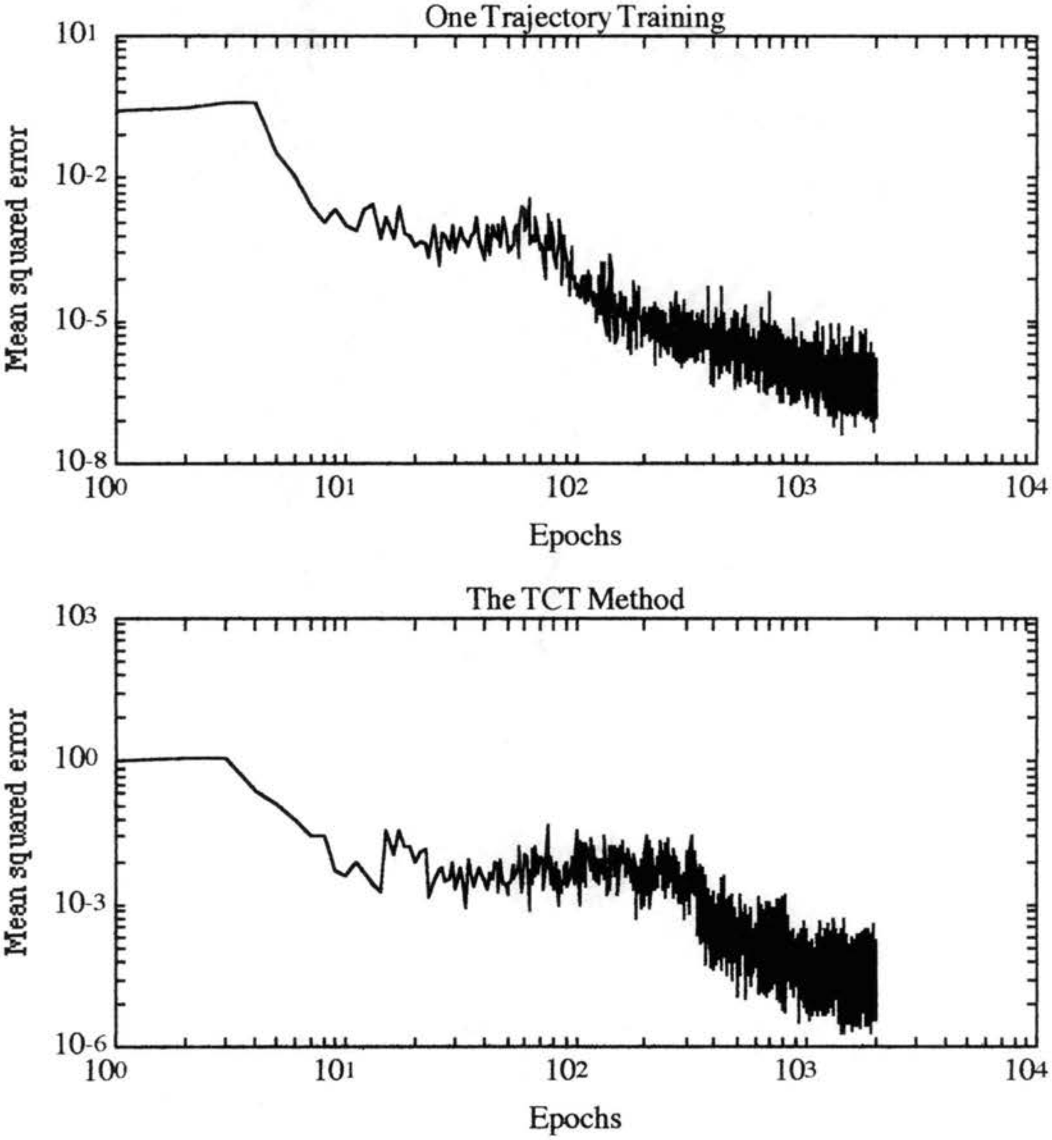


Figure 9.14 Learning Curves of One Trajectory Training vs the TCT Method for Case 2

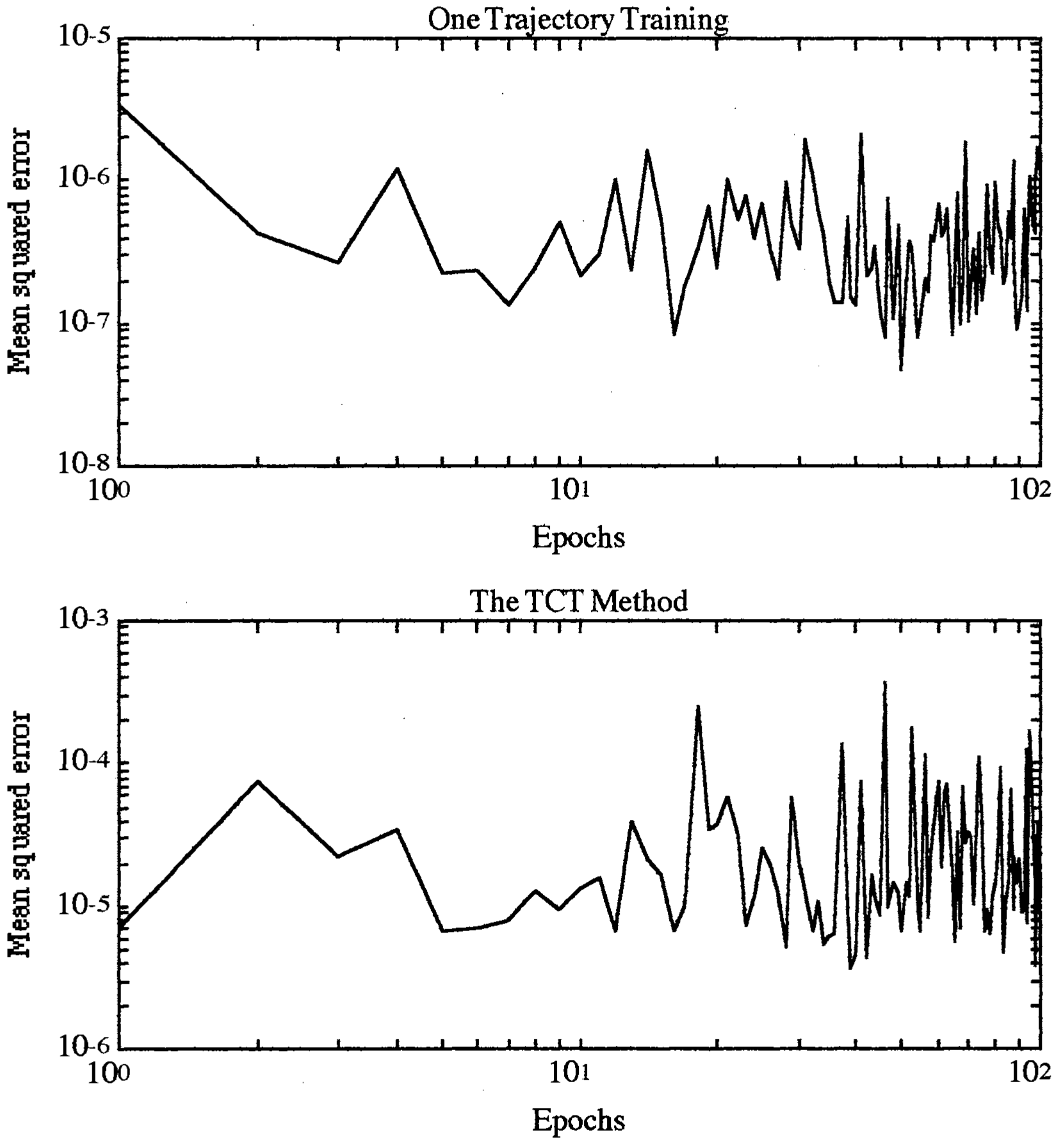


Figure 9.15 Enlargement of the Learning Curves for Epochs 1901 to 2000 from Figure 9.14

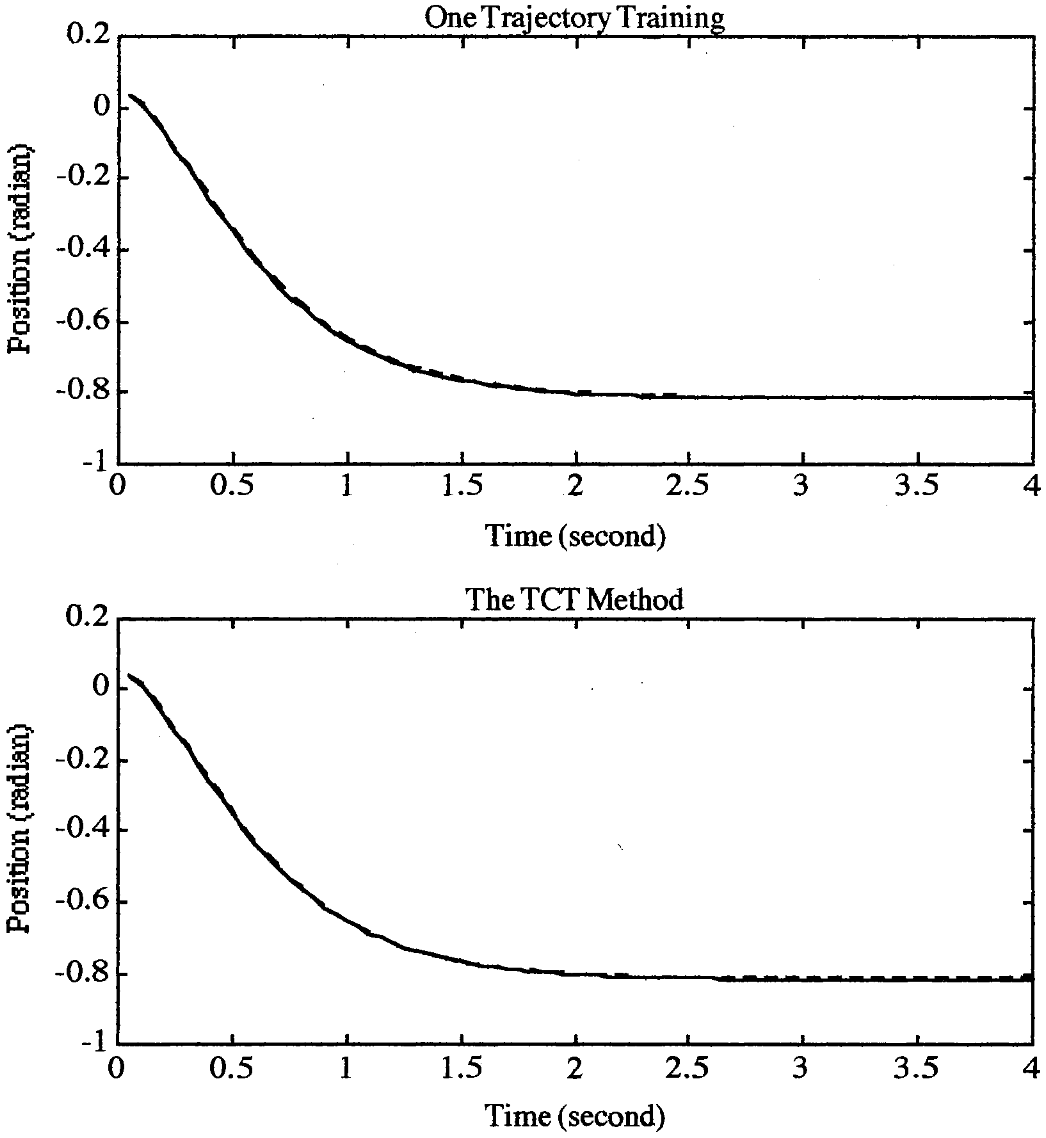


Figure 9.16 The Parallel Test Results for the Trained Controllers for Case 2

Comments

From the above simulations, it seems that the TCT method in dynamic learning only works well if the total number of training data points is small (20 points in our Case 1). Thus, this method is suited to the situation where one cannot collect a larger amount of training data. An example of this situation is Simulation 5 in Chapter 7. Since the plant in that simulation problem is a nonlinear second-order pendulum system and the initial weights that were assigned to the network controller were random, the first epoch of the learning process is composed of out of boundary (0 and π) runs. Constrained by the boundaries, each of those runs are only a few data points in length.

CHAPTER X

SUMMARY AND CONCLUSIONS

In this chapter, we will summarize the original ideas proposed by this research that appear in the discussions of the static and dynamic learning methods. We will also discuss the important results found throughout this document.

The original ideas proposed in this document can be classified into two types. The first type concerns the dynamic learning derivative method. It is discovered that the proposed true derivative equations for both the FP and BTT dynamic learning algorithms can be derived with backpropagation and the chain rule. Only the choice of the term for taking the explicit derivative in the chain rule distinguishes the two dynamic learning algorithms. More important, with the proposed derivative methods, the analysis and implementation of complex structural applications (such as the indirect MRAC architecture) can be easily achieved compared to other known approaches (like the Werbos BTT method and the Narendra dynamic backpropagation method).

The second original contribution relates to the secondary training methods used in supervised learning. In this document, the underlying algorithms for the static and dynamic learning methods are the Marquardt optimization technique using basic backpropagation and the Marquardt optimization technique using FP/BTT algorithms. We call them the primary training methods. Unfortunately, from our empirical experience, these primary training methods do not always guarantee a satisfactory training result. Therefore, so-called "secondary" training methods were proposed to aid the primary methods in training the neural networks. We have presented and implemented a static secondary learning method -- the ATS (Alternating Training data Set) method and one combined dynamic secondary learning method -- the TCT (Trajectory Cross Training) /TLI (Trajectory Length Increasing) method. Due to these proposed secondary training methods, all the computer simulations performed in Chapters 4 and 6 achieved satisfactory results. In Chapter 9, we further investigated the proposed secondary methods. We concluded that the ATS method does have better performance than the commonly used general approach. The TCT/TLI method does aid the primary method in achieving convergence.

In Chapter 2 several computer simulations were executed to investigate the effects of varying the number of neurons and the number of layers in the network when performing the function approximation task. The results suggested that a single hidden layer network is sufficient for nonlinear function approximation. The fact that all the neural networks, either feedforward or recurrent, that were successfully trained throughout this document have only a single hidden layer gives further support to this assertion.

An unexpected success, discussed in Chapter 4, came in modeling the inverse dynamics of the pendulum system. The impressive generalization ability of the trained direct inverse neurocontroller was validated during our evaluation. The controlled plant closely followed a reference model, which had never been included in the learning process. This event has revealed the potential ability of the direct inverse method in neurocontrol.

To the author's knowledge no researcher has reported successful implementation of the dynamic learning algorithm on the model reference adaptive controller. The fact that we did this makes the computer simulations performed in Chapter 6 very significant. They are the most important result of this research. In our analysis of this success, two factors contributed to this success. The first is the aid of the secondary training method, TCT/TLI, to the primary training method. The second is the Marquardt optimization method, whose fast convergence enables us to shorten the network training time and thus cut down the time consumed by the trial-and-error process. This speed is significant when compared to other optimization techniques such as were described in Chapter 3.

The final important result is described in Chapter 8. From the successes in the computer simulations, we advanced to implement the trained networks that use both static learning and dynamic learning on real physical systems in real-time. We succeeded in controlling the physical pendulum system in real-time using feedback linearization neurocontroller. We also identified the physical system using neural network modeling. Constrained by the real-time operating environment, we can only train the MRAC controller off-line in this research. Since the plant model we trained only approximates to the real plant in the pendulum moving-down movements, we thus train the MRAC neurocontroller with only the moving-down trajectories. The most important result in this research is that we trained an MRAC controller which can adequately control the real physical pendulum system in real-time. Although the off-line trained controller may be promising, however, the better way to train the MRAC controller should be on-line with the real plant. Constrained by the time frame of this research, we can only attempt to do this in the future.

REFERENCES

[1] McCulloch, W. S. & Pitts, W. (1943). "A logical calculus of the ideasimminent in nervous activity". Bulletin of Mathematical Biophysics, 5, pp 115-133.

[2] Rosenblatt, F. (1958). "The perceptron: a probabilistic model for information storage and organization in the brain". Psychological Review, 65(6), pp 386-408.

[3] Gupta, M. M. & Rao, D. H. (1993). "Dynamic Neural Units with Applications to the Control of Unknown Nonlinear Systems". Journal of Intelligent and Fuzzy Systems, 1, pp 73-92.

[4] Narendra, K. S. (1992). "Adaptive Control of Dynamical Systems Using Neural Networks". In D. A. White & D. A. Sofge (Ed.), Handbook of Intelligent Control - Neural, Fuzzy, and Adaptive Approaches (pp 141-183). New York: Van Nostrand Reinhold.

[5] Hornik, K., Stinchcombe, M. & White, H. (1989). "Multilayer Feedforward Networks are Universal Approximators". Neural Networks, 2, pp 183-192.

[6] Werbos, P. J. (1982). "Applications of Advances in Nonlinear Sensitivity Analysis". In R. Drenick & F. Fozin (Ed.), Systems Modeling and Optimization: Proceeding of 10th IFIP Conference (pp 762-770). New York: Springer-Verlag.

[7] Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986). "Learning Representations by Back-propagating Errors". Nature, 323, pp 533-536.

[8] Hagan, M. H. (1992). Tutorials from Neural Networks course.

[9] Hagan, M. H. & Menhaj, M. (1993). "Training Feedforward Networks with the Marquardt Algorithm". To be published in IEEE Transactions on Neural Networks.

[10] Narendra, K. S. & Parthasarthy, K. (1990). "Identification and Control of Dynamical Systems Using Neural Networks". IEEE Transactions on Neural Networks, 1, pp 4-27.

[11] Narendra, K. S. & Parthasarthy, K. (1991). "Gradient Methods for the Optimization of Dynamical Systems Containing Neural Networks". IEEE Transactions on Neural Networks, 2, pp 252-262.

[12] Narendra, K. S. & Parthasarthy, K. (1989). "Back Propagation in Dynamical Systems Containing Neural Networks". Technical report, Center for System Science, Yale University. New Haven, CT

[13] Narendra, K. S. & Annaswamy, A. M. (1989). <u>Stable Adaptive Systems</u>. Englewood Cliffs, NJ: Prentice Hall.

VITA 2

Wei-Chung Yang

Candidate for the Degree of

Doctor of Philosophy

Thesis: NEUROCONTROL USING DYNAMIC LEARNING

Major Field: Electrical Engineering

Biographical:

Personal Data: Born in Hong Kong, September 25th,1951, the son of Zi Yang and Wha-Sun King.

Education: Graduated from National Taipei Institute of Technology in May of 1975; Received Master of Engineering from the University of Tulsa in December 1982; Completed the requirements for the Doctor of Philosophy degree at Oklahoma State University in May 1994.

Professional Experience: Graduate Teaching Assistant, The University of Tulsa, 1981; Engineer, Burtek Inc., 1982 to 1985; Engineer, Syntech Co., 1985 to 1986; System Analyst, Taiwan Tax Bureau, 1986 to 1989; Graduate Teaching/Research Assistant, Oklahoma State University, 1990 to 1993; Post-Doctor Researcher, Oklahoma State University, 1994.

Membership in Professional Societies: Institute of Electrical and Electronics Engineers in Neural Networks, SMC, and Control Systems Societies; International Fuzzy Systems Association; International Neural Network Society.