INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 1356820

**A learning automaton approach to trajectory learning and
control system design using dynamic recurrent neural networks**

Condarcure, Thomas A., M.S.

The University of Arizona, 1993

Copyright ©1993 by Condarcure, Thomas A. All rights reserved.

U·M·I

**300 N. Zeeb Rd.
Ann Arbor, MI 48106**

A LEARNING AUTOMATON APPROACH TO TRAJECTORY LEARNING AND
CONTROL SYSTEM DESIGN USING DYNAMIC RECURRENT NEURAL NETWORKS

by
Thomas A. Condarcure

Copyright© Thomas A. Condarcure 1993

A Thesis Submitted to the Faculty of the
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE
WITH A MAJOR IN ELECTRICAL ENGINEERING

In the Graduate College
THE UNIVERSITY OF ARIZONA

1 9 9 3

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgement of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED: Thomas A. Condarcu

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

_____ Malur K. Sundareshan Professor of Electrical and Computer Engineering	12/6/93 _____ Date
--	--------------------------

Table of Contents

List of Figures	6
List of Tables	8
ABSTRACT	9
 Chapter 1 - Introduction	 10
1.0 Neural Networks in Control System Design . .	10
1.1 Review of Neural Network Models	11
1.1.1 McCulloch and Pitts Model	11
1.1.2 Hopfield Model	13
1.1.3 Perceptrons	15
1.1.3 Dynamic Recurrent Networks	19
1.2 Learning in Neural Networks	21
1.2.1 Backpropagation in Static Networks . .	22
1.2.2 Backpropagation in Dynamic Recurrent Networks	24
1.2.3 Stochastic Learning Methods	26
1.2.4 Genetic Learning	29
1.2.5 Unsupervised Learning	30
1.3 Time-Dependent Trajectory Learning	35
1.4 Application of Neural Networks to Control . .	42
1.4.1 Regulator Problems	42
1.4.1 Inverse Dynamics	45
1.4.3 Control by Identification of System	

Dynamics	46
1.4.4 Continuous Time Control	48
1.5 Organization of Thesis	49
 Chapter 2 - Learning Automata	53
2.0 Learning Automata	53
2.1 Learning Automata and Neural Networks	57
2.1.1 Learning Configuration	58
2.1.2 Learning Algorithm	59
2.1.3 Convergence	60
2.2 Incremental Learning	62
2.3 Neural Network Structure Variation	65
 Chapter 3 - Simulation	67
3.0 Simulation Structure	67
3.1 Data Structures	68
3.1.1 Neural Network Structures	68
3.1.2 Learning Automaton Data Structures	69
3.1.3 Gradient Descent Data Structures	69
3.2 Algorithms	69
3.2.1 Learning Automaton Algorithms	69
3.2.2 Gradient Descent Algorithms	72
3.3 Program Structure	75
3.4 Comparison of Methods	75
 Chapter 4 - Trajectory Learning Results	77

4.0	Simulation Environment	77
4.1	Circle Trajectory	77
4.1.1	Experiment 1	78
4.1.2	Experiment 2	81
4.1.3	Experiment 3 - A Nonlinear Reinforcement Scheme	84
4.2	Experiment 4 - Gradient Descent Learning . .	91
Chapter 5	- Neural Control	95
5.0	Control System Design	95
5.1	Control of a Linear Plant	96
5.2	Nonlinear Plant Stabilization and Control . .	102
5.2.1	Stabilization	102
5.2.2	Nonlinear Control	113
Chapter 6	- Conclusions	119
6.0	Contributions	119
6.2	Topics for Further Investigation	120
References		122

LIST OF ILLUSTRATIONS

Figure 1	McCulloch-Pitts Neuron Model	12
Figure 2	Two Layer Perceptron	16
Figure 3	A Dynamic Neuron	20
Figure 4	A Recurrent Neural Network	21
Figure 5	Architecture for Unsupervised Learning . . .	32
Figure 6	Simple Competitive Network	34
Figure 7	Inverted Pendulum	43
Figure 8	Inverse Dynamics Controller	45
Figure 9	Identification of Nonlinear Plants	47
Figure 10	Learning System	55
Figure 11	Trajectory Targets	65
Figure 12	Circle Generation	68
Figure 13	Actions per Learning Increment - Experiment	
1		79
Figure 14	Final Trajectory - Experiment 1	80
Figure 15	Final Trajectory - Experiment 4	82
Figure 16	Changes per Learning Step - Experiment 4 . .	83
Figure 17	Trajectory From Nonlinear Learning	89
Figure 18	Behavior Modifications Stored	90
Figure 19	Error Vs. Time for Gradient Learning	92
Figure 20	Toomarian & Barhen Error Graph	93
Figure 21	Final Trajectory from Gradient Learning . .	93
Figure 22	Control System Configuration	96

Figure 23	Neural Network/Plant Configuration	98
Figure 24	Generated Control Signal	99
Figure 25	Desired and Actual Outputs	100
Figure 26	Output Error	100
Figure 27	Response for Random Initialization	102
Figure 28	Error in dx/dt Versus Training Loop	105
Figure 29	Error in x Versus Training Loop	106
Figure 30	Expediency	107
Figure 31	Control Signal For $x(0) = +1$	108
Figure 32	Plant Output For $x(0) = +1$	108
Figure 33	Control Signal For $x(0) = -1$	109
Figure 34	Plant Output For $x(0) = -1$	109
Figure 35	Output for Alternate Initial Conditions	113
Figure 36	Nonlinear System Output Results	115
Figure 37	Neural Network Control Signal	115
Figure 38	Error vs. Time - Nonlinear Control	116
Figure 39	Response for Random Initialization	118

LIST OF TABLES

Table I	Comparison of Learning Methods	76
Table II	Neuron Interconnection Weights	80
Table III	Neuron Time Constants and Gains	81
Table IV	Neuron Interconnection Weights - Experiment 2	83
Table V	Time Constants and Gains - Experiment 2 . . .	84
Table VI	Neuron Interconnection Weights - Experiment 3	90
Table VII	Time Constants and Gains - Experiment 3 . .	91
Table VIII	Neuron Interconnection Weights - Control System	98
Table IX	Time Constants and Gains	99
Table X	Neuron Interconnection Weights ... first half	110
Table XI	Neuron Interconnection Weights ... second half	111
Table XII	Time Constants & Gains	112
Table XIII	Neuron Interconnection Weights 1 - Nonlinear Control System	116
Table XIV	Neuron Interconnection Weights 2 - Nonlinear Control System	117
Table XV	Time Constants and Gains - Nonlinear Control System	117

ABSTRACT

This thesis presents a method for the training of dynamic, recurrent neural networks to generate continuous-time trajectories. In the past, most methods for this type of training were based on gradient descent methods and were deterministic. The method presented here is stochastic in nature. The problem of local minima is addressed by adding the enhancement of incremental learning to the learning automaton; i.e., small learning goals are used to train the neural network from its initialized state to its final parameters for the desired response. The method is applied to the learning of a benchmark continuous-time trajectory -- the circle. Then the learning automaton approach is applied to stabilization and tracking problems for linear and nonlinear plant models, using either state or output feedback as needed.

Chapter One

Introduction

1.0 Neural Networks in Control System Design

The capabilities of control systems seem to follow the technological advances of the day. Early control systems were mechanical in design. Some examples are the flyweight controls for steam engines and simple gyroscope controls for roll correction in early airplanes. With the advent of electronics, more sophisticated controls are realized, and the design of these controllers has evolved from single-input, single-output, linear designs to multi-input, multi-output, linear designs and onward to more complex distributed and nonlinear controllers. Digital electronics continued the evolution, but limitations in these design techniques always forced the designer to make compromises.

Continuing this trend, another technique to apply to the design of controllers is the application of neural networks. These networks are composed of interconnections of artificial neurons -- circuit elements which simulate some

of the more important properties of biological neurons. The promise of this approach is the possibility of a massively parallel computation capability and adaptability to changing environmental conditions. However, there are several problems that have yet to be solved in order to make this approach practical. Among these are the difficulty of training large networks, training for true continuous-time trajectory generation and determining an optimal network size; i.e., the number of neurons.

The remainder of this section will be a brief overview of several important aspects of neural networks, including neuron models, network architecture and training. This summary must necessarily be brief; further information can be found in the references cited. The focus of this overview will be the possible application of neural networks for continuous-time control system design using existing training methods. These training methods can either be supervised, where an error criterion is explicitly minimized, or unsupervised, where the learning does not make explicit use of error functionals or training feedback in the calculation of the modifications to be made to the network.

1.1 Review of Neural Network Models

1.1.1 McCulloch and Pitts Model

The earliest form of neuron modelling is found in the

work of McCulloch and Pitts [1]. This simplified model, shown in Figure 1, is a binary threshold unit; that is, the unit is capable of a binary output based on the weighted sum of the inputs with respect to some threshold level. The weighting of the inputs corresponds to the nature of the synapses in a biological network; a positive weight represents an excitatory synapse while a negative weight represents an inhibitory synapse.

Figure 1 McCulloch-Pitts Neuron Model

A mathematical representation for this neuron model is given by

$$n_i(t+1) = \theta \left(\sum_{j=0}^N (w_{ij}n_j(t) - \mu_i) \right) \quad (1)$$

where

$$\theta(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

McCulloch and Pitts proved [1] then when the weights are chosen correctly and the network is updated

synchronously, networks of these types of neurons are capable of universal computation.

1.1.2 Hopfield Model

A similar model of the neuron was used by Hopfield [3] to formulate the concept of *associative memory*. This neuron model is binary as in the case of the McCulloch and Pitts model. The neurons are connected in a one-layer network, that is, each neuron is connected to every other neuron. If some neurons are designated as output neurons, then there is feedback from these output neurons to the other neurons. For a given N-unit network, where N denotes the number of neurons and ξ_i denotes the output of the *i*th neuron, it is desired to output a specific pattern of outputs for a certain class of inputs. This class of inputs might be all those that are "close" to the desired pattern, where a metric must be defined in order to determine "closeness".

For the simple case of learning one pattern, it is only necessary that the output of the network be stable when it has attained the desired pattern

$$\text{sgn}(\sum_j w_{ij}\xi_j) = \xi_i \quad , \quad (3)$$

where ξ_i is the desired pattern and ξ_j is the starting value of the outputs of the network. Note that the signum function used in equation (3) is defined by

$$\text{sgn}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases} . \quad (4)$$

Because the outputs and inputs can take only the values +1 and -1, it is seen that

$$\xi_j^2 = 1 . \quad (5)$$

So when

$$w_{ij} \propto \xi_i \xi_j , \quad (6)$$

and this is substituted into equation (3), the desired pattern is achieved whenever more than half of the bits are correct. If some of the neurons are not in the correct state initially, the correct bits will force the state of the network to the desired pattern. Consider the sum variable h_i where

$$h_i = \sum_j w_{ij} \xi_j . \quad (7)$$

It is seen that the correct bits will overwhelm the sum and force each bit to the correct state. In other words, once the weights have been selected for the desired pattern, that pattern becomes an *attractor* for the network. However, it is also true that the opposite state (where each output is of the opposite polarity from the desired polarity), called the *reversed state*, is also an attractor by equation (5); i.e., when $\xi_j = -1$, $\xi_j^2 = 1$ and if the state of the network is at $\xi_j = -1$, it will stay there, as can be seen when these values are substituted into equation (3).

The *generalized Hebb rule* for learning can be used where there is more than one pattern to be learned. This rule will be defined below. The learning rule defined in equation (6) is a form of Hebbian learning.

This, then, is the *associative memory*. A given set of desired patterns can be learned by the neural network, and a particular pattern will be retrieved when an input pattern is "close" enough to it.

Hopfield also defined [3] the concept of the *energy function*. Since each desired state is an attractor, it is a point of minimum energy of the system as the state evolves according to its dynamic rule. Hopfield was able to show that the weights determined by using Hebbian learning are the optimum weights for minimizing the energy function. The function itself is given by

$$H = -\frac{1}{2} \sum_{ij} w_{ij} S_i S_j \quad (8)$$

where S_i is the desired state and S_j is the actual state.

1.1.3 Perceptrons

Rosenblatt [2] worked with networks of McCulloch-Pitts neurons, primarily trying to determine ways of calculating the weights, w_{ij} . The network architecture he proposed consisted of a layered, feed-forward configuration called the *perceptron*. Again, the neuron has the mathematical representation

$$S_i = \text{sgn}(\sum_j w_{ij} S_j - \theta_i) \quad . \quad (9)$$

The threshold of each unit, θ_i , is sometimes dropped or set equal to zero. Very similar networks called *adalines* were invented around the same time by Widrow and Hoff [48], where *adaline* is an acronym for *adaptive linear estimator*. The *adaline* is also a feedforward network of static neurons used for control system applications.

Figure 2 Two Layer Perceptron

Rosenblatt was able to show the convergence of a learning algorithm for a simple class of perceptrons that did not have any intermediate layers. However, Minsky and Papert [47] were able to show that this simple type of perceptron was incapable of solving even very simple problems; the most famous of these is the XOR problem. Although this stifled further research for the next 20 years, it was later shown that the XOR problem and others could be solved by multilayer networks.

It has been shown that in order for a problem to be

solvable by a perceptron, the problem must be *linearly separable*. Referring to Figure 2, it is seen that the perceptron has N inputs and M outputs, and these are defined as either ± 1 or 0 and 1 . The input can be considered an N -dimensional vector, called a *pattern vector*, and designated ξ^μ . The outputs can also be considered vectors in an M -dimensional space and are designated O^i . For each pattern vector, there is an associated target vector, ζ^i in M -space, which is the desired output for that pattern vector. Linear separability means that a plane can be found in the ξ space such that the patterns characterized by $\zeta^\mu = +1$ can be separated by that plane from the $\xi^\mu = -1$ patterns. The XOR problem is not linearly separable for a simple one-layer perceptron.

A simple learning rule for perceptrons is based on Hebbian learning. The "Hebb" rule makes each weight the superposition of terms of the form

$$w_{ij} = \frac{1}{N} \xi_i \xi_j \quad (10)$$

where N is the total number of units in the network. This gives the learning equation for the weights

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^P \xi_i^\mu \xi_j^\mu \quad (11)$$

where the term, ξ_i^μ , denotes the i^{th} bit of training pattern μ .

The training algorithm eventually developed by Rosenblatt for the perceptron weights is given by

$$w_{ij}^{new} = w_{ij}^{old} + \Delta w_{ik} \quad (12)$$

where

$$\Delta w_{ik} = \begin{cases} 2\eta \zeta_i^\mu \xi_k^\mu & \text{if } \zeta_i^\mu \neq O_i^\mu \\ 0 & \text{otherwise} \end{cases} \quad (13)$$

Note that this rule applies when the outputs only take on the values ± 1 . The parameter, η , is called the *learning rate*.

If a problem has been shown to be linearly separable (otherwise a solution does not exist), it can be shown that the dot product of the desired and the actual outputs must be ± 1 , or $\zeta_i^\mu O_{i\mu} = \pm 1$. Splitting up equation (13) into two equal terms gives

$$\Delta w_{ik} = \eta (1 - \zeta_i^\mu O_{i\mu}) \zeta_i^\mu \xi_k^\mu \quad (14)$$

or

$$\Delta w_{ik} = \eta (\zeta_i^\mu - O_{i\mu}) \xi_k^\mu \quad (15)$$

Let

$$h_i^\mu = \sum_k w_{ik} \xi_k^\mu \quad (16)$$

which is then the input to the $\text{sgn}()$ function. It may be desirable to have this value be greater than some threshold value as well as have the correct sign, which modifies the product term in the learning rule to

$$\zeta_i^\mu h_i^\mu \equiv \zeta_i^\mu \sum_k w_{ik} \xi_k^\mu > N\kappa \quad , \quad (17)$$

where the product, $N\kappa$, defines the desired threshold in terms of the size of the network, where this threshold gives some measure of confidence that switching will occur as desired. In a hardware implementation of these networks, this threshold will help to overcome noise problems which cause spurious switching.

The sum over k scales with N , and hence, in order to keep the margin size, κ , fixed for any number of input units, N is included on the right-hand side. This changes the learning rule to

$$\Delta w_{ik} = \eta \Theta(N\kappa - \zeta_i^\mu h_i^\mu) \zeta_i^\mu \xi_k^\mu \quad (18)$$

where Θ is the unit step function.

It should be noted that in these cases, for Hebbian learning to be valid, it is assumed that $w_{ij} = w_{ji}$.

1.1.3 Dynamic Recurrent Networks

The neuron models described in the previous sections may all be classified as static models; that is, for a particular input, there is an instantaneous change in the neuron to a new state. However other models may be defined where the state of the neuron evolves over time with respect to its internal state and the state of the inputs. Models such as these would define a *dynamic* neuron. One possible form for a dynamic model given by Cohen and Grossberg [53]

and Hopfield [54] is

$$\frac{dV_i}{dt} + \tau_i V_i = \tau_i g\left(\sum_j w_{ij} V_j + \xi_i\right) \quad , \quad (19)$$

where V_i denotes the state of the i^{th} neuron in a particular network, τ_i is its time constant called the *relaxation time*, $g()$ is some nonlinear function, normally a sigmoid function such as those shown in equation (20) and equation (21) below. ξ_i is some threshold value for switching. A dynamic neuron is shown in Figure 3.

Figure 3 A Dynamic Neuron

Now the neuron output is a continuous function of time, dependent on its initial state and the state of the other

neurons in the network.

Networks, such as the perceptron, are characterized by having fixed layers of neurons connected in a feedforward configuration. Another possible configuration is to allow connections in the feedback direction. When connections are allowed in both the feedforward and feedback directions, the network is said to be recurrent. An example of a recurrent network is shown in Figure 4.

Figure 4 A Recurrent Neural Network

Learning methods for dynamic, recurrent networks are somewhat more complicated than those for static feedforward networks. Some of these methods are described in the next section. A new training method will be described in Chapter 2.

1.2 Learning in Neural Networks

Several learning methods have been described in the previous paragraphs as they relate to specific neuron models and network configurations. Other learning methods have also been developed which can be applied to these same models.

1.2.1 Backpropagation in Static Networks

Backpropagation is a method that has been successfully used for training several different configurations of neural networks. It was invented independently by Werbos [5], Parker [6] and Rumelhart, Hinton and Williams [7]. This is a gradient scheme and, as such, requires that the mathematical description of the neuron nonlinearity be differentiable. Consequently, the McCulloch-Pitts model can no longer be used.

Several different kinds of nonlinear input-output functions have been used for the neuron. Examples of these are

$$g(h) = \tanh(\beta h) \equiv \frac{e^{\beta h} - e^{-\beta h}}{e^{\beta h} + e^{-\beta h}} \quad (20)$$

and

$$g(h) = f_{\beta}(h) \equiv \frac{1}{1 + e^{-2\beta h}} \quad (21)$$

Backpropagation to determine the weights for a feedforward network as shown in Figure 2 is comprised of several steps:

1. The weights are initialized to small, random values.
2. An input pattern, ξ_k^{μ} , is applied to the neural

network input layer.

3. The signals are propagated forward from the input layer through the network to the output layer via the neuron equation

$$V_i^M = g\left(\sum_j w_{ij} V_j^{M-1}\right) \quad . \quad (22)$$

where M denotes the particular later under consideration and i denotes the i^{th} neuron in that layer.

4. A value that will be used to calculate the change in weights is determined by taking the gradient of the error criterion. In the case where this is the square of the difference between the input and the output, this value is

$$\delta_i^M = \frac{\partial E}{\partial w_{ij}} = g'(h_i^M) [\zeta_i^M - V_i^M] \quad (23)$$

where $g'()$ denotes the derivative of the nonlinearity with respect to the weight and h_i^M is the summation shown in equation (22).

5. The gradient can be calculated for weights in each previous layer by using a similar method. Taking the error for the output layer and substituting into equation (22) for V_i^M gives

$$\delta_i^{M-1} = g'(h_i^{M-1}) \sum_j w_{ij}^M \delta_j^M \quad . \quad (24)$$

This step propagates the error backward through the

system for the gradients, or the *backpropagation* step.

6. The change in the weights is then given by

$$\Delta w_{ij}^M = \eta \delta_i^M V_j^{M-1} \quad . \quad (25)$$

The weights are then updated by

$$w_{ij}^{new} = w_{ij}^{old} + \Delta w_{ij} \quad . \quad (26)$$

7. These steps are repeated for the succeeding layers by propagating the error backwards using

$$\delta_i^{m-1} = g'(h_i^{m-1}) \sum_j w_{ji}^m \delta_j^m \quad (27)$$

for $m = M, M-1, \dots, 2$ until a delta has been calculated for every unit and all units have been updated according to equation (12).

8. The process is repeated for the remaining training patterns.

1.2.2 Backpropagation in Dynamic Recurrent Networks

Backpropagation learning can be extended to dynamic recurrent networks. This has been independently pointed out by Pineda [8], Almeida [9] and Rowher and Forrest [10]. Manipulating equation (19) gives

$$\tau_i^{-1} \frac{dV_i}{dt} = -V_i + g\left(\sum_j w_{ij} V_j + \xi_j\right) \quad . \quad (28)$$

It is assumed that only stable responses of the neural network are desired. This condition reduces to

$$\frac{dV_i}{dt} = 0 \quad . \quad (29)$$

With this requirement, equation (28) reduces to

$$V_i = g(\sum_j w_{ij} V_j + \xi_i) \quad (30)$$

which is the same condition as for the static neuron.

There are several error criteria that can be used to determine the gradients necessary for backpropagation. The integral of the square of the error is the easiest to work with since it is differentiable. This is given by

$$error = E = \sum_{i \in \Omega} (\zeta_i - V_i)^2 \quad . \quad (31)$$

where Ω is the set of output nodes.

The change in the weights is determined from

$$\Delta w_{pq} = -\eta \frac{\partial E}{\partial w_{pq}} = \eta \sum_k E_k \frac{\partial V_k}{\partial w_{pq}} \quad (32)$$

where E_k is the error for output unit k . The partial derivative on the right hand side of equation (32) is determined by differentiating the fixed-point equation, equation (30), with respect to the weight, w_{pq} . This partial derivative becomes

$$\frac{\partial V_i}{\partial w_{pq}} = g'(h_i) [\delta_{ip} V_q + \sum_j w_{ij} \frac{\partial V_j}{\partial w_{pq}}] \quad (33)$$

where

$$h_i = \sum_j w_{ij} V_j + \xi_i \quad (34)$$

and δ_{ip} is the Kronecker delta function. A set of linear equations can be generated by taking the partial derivative for all the outputs with respect to the weight under consideration. This system of equations can then be solved to determine the desired partial derivatives to substitute into the right hand side of equation (32). Then the desired Δw_{pq} can be calculated.

Note that in order for this method to work, the system of equations must be consistent; otherwise, the desired partial derivatives cannot be exactly determined.

1.2.3 Stochastic Learning Methods

The learning methods described in the preceding paragraphs might be termed deterministic methods as they use the mathematical description of the neurons explicitly in the determination of the changes to the weights. However, there are other neuron models and training methods that are based on stochastic and statistical arguments.

One such model, which has been used by Hinton and Sejnowski [11] and Peretto [12] considers the neural network to be made up of a lattice of binary units where the state of each unit is determined by

$$h_i = \sum_j w_{ij} S_j + h^{ext} \quad (35)$$

where h_i is the state of neuron i , w_{ij} measures the effect

of neuron j on neuron i and h^{ext} is some influence external to the neural network.

The motivation for this approach is taken from the field of statistical mechanics, related specifically to magnetic materials. The state of a neuron, S_i , is likened to the state of a magnetic particle which can have one of only two orientations, $+1$ or -1 . The external influence corresponds to some external magnetic field.

So-called "thermal" effects are then added into the model. These effects take into account the stochastic nature of the transitions of the neuron outputs. In other words, the deterministic rule for neuron transitions becomes stochastic with the degree of randomness to be determined by a parameter called "temperature". At any particular time,

$$S_i = \begin{cases} +1 & \text{with probability } f(h_i) \\ -1 & \text{with probability } 1 - f(h_i) \end{cases} \quad (36)$$

The choice for the probability density function is given by

$$g(h) \equiv f_\beta(h) = \frac{1}{1 + \exp(-2\beta h)} \quad (37)$$

The parameter β is related to the "absolute temperature" T (desired randomness) by

$$\beta = \frac{1}{\kappa T} \quad (38)$$

where κ is Boltzmann's constant. Note that as $T \rightarrow 0$, the unit becomes deterministic.

Using equation (36), an average value of the state of each neuron can be defined as

$$\begin{aligned}\langle S \rangle &= (+1) \cdot \text{Prob}(S = +1) + (-1) \cdot \text{Prob}(S = -1) \\ &= \frac{1}{1 + \exp(-2\beta h)} - \frac{1}{1 + \exp(2\beta h)},\end{aligned}\quad (39)$$

where the notation $\langle \rangle$ denotes the statistical average. This reduces to

$$\langle S \rangle = \tanh(\beta h) \quad . \quad (40)$$

An enhancement of this method of analysis is the method of *mean field theory*, where the true value of h_i is replaced by an average value

$$\langle h_i \rangle = \sum_j w_{ij} \langle S_j \rangle + h^{ext} \quad (41)$$

where the average output, $\langle S_i \rangle$, can be computed from

$$\langle S_i \rangle = \tanh(\beta \langle h_i \rangle) = \tanh(\beta \sum_j w_{ij} \langle S_j \rangle + \beta h^{ext}) \quad . \quad (42)$$

By applying the equations for Hebbian learning to equation 41, the weights for this network can be calculated using

$$\langle S_i \rangle = \tanh\left(\frac{\beta}{N} \sum_{j,\mu} \xi_i^\mu \xi_j^\mu \langle S_j \rangle\right) \quad . \quad (43)$$

In this mean field method, it is not so much the training method as the neuron model which is stochastic.

One stochastic learning method, based on the so-called ALOPEX methods, has been described by Venugopal, Pandya and Sudhakar [13]. This method is independent of the neuron model used.

With this method, during the n^{th} iteration, the weight, w_{ij} , is updated through

$$w_{ij}(n) = w_{ij}(n-1) + \delta_{ij}(n) \quad (44)$$

where δ_{ij} is a small negative or positive step of size δ . The polarity of the increment is determined from the probabilities

$$\delta_{ij}(n) = \begin{cases} -\delta & \text{with probability } P_{ij}(n) \\ +\delta & \text{with probability } 1 - P_{ij}(n) \end{cases} \quad (45)$$

where

$$P_{ij}(n) = \frac{1}{1 + \exp\left(\frac{-\Delta_{ij}(n)}{T}\right)} \quad (46)$$

The parameter, $\Delta_{ij}(n)$, is the correlation between the change in weight, w_{ij} , and the associated change in the error functional defined for learning; e.g., the sum of the square of the errors in the output nodes. The parameter, T , is started out at a large value and, as learning progresses, it is set to the average value of the correlation. With a large value of T , it is seen that the probability, $P_{ij}(n)$, starts out at 0.5 for $n = 0$. As the correlation between a change and its corresponding increase or reduction in error progresses, the selection of that polarity for the change in weight will decrease or increase, respectively.

1.2.4 Genetic Learning

Genetic learning is a method that attempts to duplicate

the process of evolution or selection of characteristics using analogs of reproduction. With this approach, the network is thought of as an entire organism whose parameters, such as neuron interconnection weights, can be associated with determining factors similar to the genes of living organisms. These individual genes can be collected into something like a chromosome that determines the overall make-up of the parameters of the network.

As in genetics, traits from one network may be combined with traits from another network, as in sexual reproduction, to form a new neural network. Mutations are also allowed based on some randomly determined rate. The resulting networks may then be subjected to some survival criteria to determine which networks may go on to produce more networks and which networks will not. In this way, a population of neural networks evolves until a population of networks is achieved which is better suited to meet the survival criteria of its environment.

The survival criteria are based upon some error which is desired to be minimized; such as, to solve some logic problem like the XOR problem, or perhaps perform pattern recognition. Ichikawa and Sawa [33] have successfully demonstrated the use of a genetic algorithm in designing controllers for linear and nonlinear systems using static networks.

1.2.5 Unsupervised Learning

Unsupervised learning methods are useful when it is desired that the neural network sort data into equivalence classes; i.e., perform a transformation from some N -dimensional input space to an M -dimensional output space where $M < N$. This method can also train networks to do encoding of data patterns of N bits into patterns of M bits, thus doing data compression. This method has also been used for teaching a network to distinguish features; for example, mapping a 2-dimensional visual array into features such as edges and simple shapes. Early research in unsupervised learning was done by Rosenblatt [2]; later researchers include von der Marlsburg [49], Fukushima [50], Grossberg [51] and Kohonen [52].

Hebbian learning can be applied for training linear units. Figure 5 shows a simple network of one linear unit. The output is just the weighted sum of the input bits. Hebbian learning has also been applied in an unsupervised environment with nonlinear units.

A simple problem for this network to learn is to find which input pattern occurs with the greatest frequency. In this case, it is desired that the more frequently a particular input vector, ξ , occurs with respect to other input patterns to the neuron, then the greater will be the value of the neuron output.

Figure 5 Architecture for Unsupervised Learning

A simple application of Hebbian learning would be to update the weights by

$$\Delta w_i = \eta V \xi_i \quad . \quad (47)$$

where V is the neuron output. This strengthens the output pattern for each input presented, so frequent inputs will have the most influence over time. However, there is no bound on the weights and the learning is never completed. Note that this learning is performed on a one-layer, feedforward network.

This problem is overcome by using Oja's rule [19] which is an enhancement of Hebbian learning. Oja's rule uses the equation

$$\Delta w_i = \eta V (\xi_j - V w_i) \quad . \quad (48)$$

This method forces the norm of the weights to equal zero.

Oja's rule for M output units is

$$\Delta w_{ij} = \eta V_i (\xi_j - \sum_{k=1}^N V_k w_{kj}) , \quad j = 1, 2, \dots, M \quad . \quad (49)$$

Another learning rule for an M-output network is Sanger's rule [20]:

$$\Delta w_{ij} = \eta V_i (\xi_j - \sum_{k=1}^i V_k w_{kj}) , \quad j = 1, 2, \dots, M \quad . \quad (50)$$

Both learning rules project the input vector ξ onto the first M *principal components* since the weights converge to an orthogonal set of unit vectors. The difference between Sanger's rule and Oja's rule is that Sanger's rule takes the principal components of the input data space in order while there is no ordering when Oja's rule is used. The principal components are the orthogonal set of axes which give a basis for the input data space into which the input data can be categorized. The selection of which rule to use for learning would depend on the desired output specifications.

Linsker [21] has successfully used these methods with multi-layer, linear arrays interfacing with 2-dimensional visual arrays. The layers were trained separately to recognize features. It may be thought that since linear elements are being used, only one transformation is needed and hence only one layer. However, by training the layers separately, simple visual images were successively refined.

With the Hebbian learning approach, the outputs can

take on any real value, and with multiple output networks, all outputs can be active to some extent. When *competitive learning* is used, only one output is active at any one time. Here competitive refers to the winner-take-all aspect of the output activation, or they compete for the one to be active. This method is useful for categorizing input data into specific categories. In order to guarantee that only one output is active at a time, *lateral inhibitory connections* are made between the output units as shown in Figure 6.

Figure 6 Simple Competitive Network

The *Standard Competitive Learning Rule* defined by Grossberg [51] is given by

$$\Delta w_{i \star j} = \eta (\xi_j^\mu - w_{i \star j}) \quad (67)$$

where $w_{i \star j}$ is the weight to unit i , when unit i is the winning unit, from unit j . $\Delta w_{i \star j} = 0$ when unit i is not the winning unit. This rule normalizes the sum of the weights to 1 and keeps the learning from becoming unbounded in a way similar to Oja's rule.

1.3 Time-Dependent Trajectory Learning

In the preceding paragraphs, the training to be accomplished with the neural networks was fixed-point learning; that is, for a fixed input vector, it is desired that there be a fixed output vector, in other words, the neural network reach some fixed stable after a finite period of time.

For applications such as pattern recognition, this is certainly adequate, since it is only required that particular patterns be determined or categorized. For control system applications, this type of network can be used for system identification, as has been shown by Narendra and Parthasarathy [14] and Karakasoglu, Sudharsanan and Sundareshan [15] and [55].

However it may also be desirable for a dynamic neural network to be able to learn to generate a control signal in continuous time. This is a time-dependent task where the states that the neural network passes through may be significant for the task of minimum energy optimization. This ability to train a dynamic, recurrent neural network to perform continuous-time trajectories is also important for control tasks where a digital control system may be difficult to implement due to instability, the loss of high frequencies or quantization errors that may be introduced due to sampling. Continuous-time trajectories are also important for the control of repetitive tasks; that is,

where a signal may have to be constantly regenerated, for example in assembly line tasks.

Some work has recently been done in the area of time-dependent trajectory learning. These efforts have used backpropagation in one form or another.

The earliest work in this area was done by Williams and Zipser [16]. The neural network used was a discrete time system using dynamic neurons as defined in section 1.4. An error functional was defined as

$$E_k(t) = \begin{cases} \zeta_k(t) - V_k(t) & \text{if } \zeta_k \text{ is defined for time } t \\ 0 & \text{otherwise} \end{cases} \quad (52)$$

where E_k is the error for output unit k . This gives a total cost function of

$$E = \sum_{t=0}^T E(t) \quad (53)$$

where

$$E(t) = \frac{1}{2} \sum_k [E_k(t)]^2 \quad (54)$$

Note that this is a discrete time method so a summation over T is used instead of an integration. Gradient descent can then be determined by taking, as in the fixed-point case,

$$\Delta w_{pq} = -\eta \frac{\partial E(t)}{\partial w_{pq}} = -\eta \sum_k E_k(t) \frac{\partial V_k(t)}{\partial w_{pq}} \quad (55)$$

The full change in w_{pq} can be taken as the sum of Δw_{pq} over T , the trajectory length. The partial derivative on the

right hand side of equation (49) can be determined in a manner similar to the fixed point case:

$$\frac{\partial V_i(t)}{\partial w_{pq}} = g'(h_i(t-1)) \left[\delta_{ip} V_q(t-1) + \sum_j w_{ij} \frac{\partial V_j(t-1)}{\partial w_{pq}} \right] . \quad (56)$$

Williams and Zipser found that updating the weights at each time increment instead of waiting until the sequence is complete works well if the learning rate, η , is sufficiently small. They have called this method *Real Time Recurrent Learning* (RTRL).

A modification they and others have found useful is called *teacher forcing*. Note that the networks used are recurrent networks, so the signals from the output neurons can be made available to all other neurons in the network. For the teacher forcing the desired output signals are propagated through the network while the actual outputs are used for calculating the error and determining the gradients. This seems to speed up the learning since, even at the beginning of learning, the actual outputs will be forced to be somewhat similar to the desired outputs, since there is some input-output relation for the neuron. Learning then progresses by forcing the actual outputs closer to the desired outputs. This is in contrast to learning without teacher forcing where the shape of the desired trajectory must be learned as well as reducing the error.

Another backpropagation method has been developed by

Pearlmutter [17]. While the method of Williams and Zipser only performs backpropagation of the weights, Pearlmutter also calculates the gradients on the time constants; i.e., the τ_i of equation (19). This method uses ordered derivatives, with variables ordered by time. The ordered derivative measures how much a small change in a neuron output at some time t affects the error measure as the change is propagated forward in time. A new variable, $z(t)$, is set equal to this ordered derivative. For an infinitesimal change to a weight, there will be a corresponding change to the error which is given by (for first order integration)

$$dE = V_i(t) g'(h_j(t)) \frac{\Delta t}{T_j} z_j(t) dw_{ij} \quad (57)$$

and is then used to calculate the gradients via

$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{\tau_j} \int_{t_0}^{t_f} V_i g'(\sum_j w_{ij} V_j) z_j dt \quad (58)$$

and

$$\frac{\partial E}{\partial \tau_i} = -\frac{1}{\tau_i} \int_{t_0}^{t_f} z_i \frac{dV_i}{dt} dt \quad (59)$$

These gradients are then used, as before, to calculate incremental changes in the weights and the time constants.

A third gradient method has been developed by Toomarian and Barhen [18]. This method uses an adjoint system of equations in conjunction with a different method of teacher

forcing to program time dependent trajectories. The adjoint equations are derived from a system of forward sensitivity equations (FSE's) which are generated by differentiating the neuron dynamics with respect to a parameter of the neural network; i.e., a weight, a time constant or a gain in the nonlinearity. Also, a modified neuron dynamic equation is used:

$$\frac{dV_i}{dt} + \tau_i V_i = g(g_i (\sum_j w_{ij} V_j + I_i)) \quad (60)$$

where the gain, g_i , controls the slope of the nonlinearity and I_i is an internal input which includes the teacher forcing. Note that $g(\cdot)$ is typically a sigmoidal function.

The teacher forcing in this method is only applied to the output neurons as an extra input. It has the form

$$I_i = \lambda (a_i(t))^{1-\beta} (a_i(t) - V_i(t))^\beta \quad (61)$$

where $a_i(t)$ is the desired value for the i^{th} output neuron, V_i is the output of the i^{th} output neuron and β is a value determined heuristically as 7/9. The value λ is a coefficient that is reduced as the error decreases in order to diminish the effect of the teacher forcing with time.

The FSE's have the form

$$\frac{d\dot{V}_I}{d\rho_i} + \sum_m A_{nm} \frac{\partial V_i}{\partial \rho_i} = S_{i,\rho} \quad (62)$$

where $\dot{V}_I = dV/dt$ and equation (60) has been differentiated

with respect to ρ_i , one of the neural network parameters,

$$A_{i,\rho_i} = \left(\tau_i - g_i g' \frac{\partial I_i}{\partial V_i} \right) \delta_{ij} - g_i g' w_{ij} \quad (63)$$

and

$$S_{i,\rho_\mu} = -V_i \delta_{\rho_\mu, x_n} + g_i g' \sum_j V_j S_{ij} \delta_{\rho_\mu, w_{ij}} + g' \delta_{\rho_\mu, g_i} \quad (64)$$

Note that $\delta_{i,j}$ is the Kronecker delta function and g' is the derivative of the sigmoidal function with respect to one of the neuron outputs in the summation of equation (60).

Solving for the gradients is accomplished by means of the adjoint system of equations (ASE's). They have the form

$$-\dot{v}_i + \sum_j A_{ij}^T v_j = S_i^* \quad (65)$$

where v_i is the adjoint variable and the superscript T denotes transpose. The source term, S_i^* , and the initial conditions need to be specified before the ASE's can be solved.

The source term of equation (65) is selected so that its inner product with one of the terms, called the *indirect effect*, in the gradient of the error with respect to the system parameter under consideration is equal to zero. This indirect effect is developed as follows. Let

$$E = \int_{t_0}^{t_f} F dt \quad (66)$$

be the error functional where F is some function like the

sum of the square of the errors. Then the gradient becomes

$$\frac{\partial E}{\partial \rho_\mu} = \int_{t_0}^{t_f} \frac{\partial F}{\partial \rho_\mu} dt \quad (67)$$

which is expanded, using the chain rule, to

$$\frac{\partial E}{\partial \rho_\mu} = \int_{t_0}^{t_f} \frac{\partial F}{\partial \rho_\mu} dt + \int_{t_0}^{t_f} \frac{\partial F}{\partial V} \cdot \frac{\partial V}{\partial \rho_\mu} dt \quad (68)$$

The second term on the right hand side is the indirect effect.

Once the indirect effect is eliminated by finding the desired source term, the gradient can be determined from combining equations to give

$$\frac{\partial E}{\partial \rho_\mu} = \int_{t_0}^{t_f} \bar{V}^T \bar{S}_{,\mu} dt - [\bar{V}^T \bar{u}_{,\mu}]_{t_f} \quad (69)$$

where the second term on the RHS is an inner product and can be determined indirectly using a simple forward integral. Note that $\bar{S}_{,\mu}$ denotes the inhomogeneous source term of the forward sensitivity equations (62) and $\bar{u}_{,\mu}$ is the sensitivity of u to the parameter μ . The bar denotes a vector, and the source term in the integral is taken as the vector of source terms determined previously.

All of these backpropagation methods suffer from the shortcoming that the desired output of the neural network must be known *a priori* in order for training to be accomplished, especially where teacher forcing is a

necessity for successful learning. This may be unrealistic in the cases where it is desired to generate a control signal for some system where that control is unknown or is difficult or impossible to calculate analytically.

1.4 Application of Neural Networks to Control

There are several approaches that may be taken when selecting a system configuration in order to meet control system specifications. Classical approaches generally revolve around the use of feedback and compensators or simply the generation, in a feedforward configuration, of a predetermined control signal, while more modern approaches typically involve the use of adaptive controllers, model reference controllers and parameter estimators. Neural networks have been used in some of these applications, both as regulators and as tracking controllers.

1.4.1 Regulator Problems

A regulator may be defined as a control system which drives a plant from some arbitrary initial state to the zero state in some finite time. The classic nonlinear regulator problem is the balancing of an inverted pendulum. In this problem, it is desired to force an inherently unstable system; i.e., vertical rod to be balanced, by applying a force in one dimension. This is shown in Figure 7. The system is considered to be stable when the rod is completely vertical and unmoving.

Figure 7 Inverted Pendulum

Using two-layer, static networks, Widrow and Smith [36] and Widrow [37] have demonstrated the ability of a neural network to regulate the inverted pendulum by using classical techniques to determine the desired control signal (the force applied to the cart) and then train a static network to simulate the control. The control signal was generated to ensure that the rod was held in the vertical position.

Barto, Sutton and Anderson [38] have also designed a controller for this problem using a two-layer network and reinforcement learning. In their configuration, the force is of a fixed magnitude.

A reinforcement scheme is used by Barto, et. al. to train a static neural network. When learning is accomplished by a reinforcement scheme, a neural network parameter is modified based on some signal coming to the neural network

from the environment.

For the Barto method, the reinforcement algorithm takes the form

$$w_i(t+1) = w_i(t) + \alpha r(t) e_i(t) \quad , \quad (70)$$

where α is a positive constant, $r(t)$ is the real-valued reinforcement at time t and $e_i(t)$ is the *eligibility*.

In this reinforcement scheme, the state space of the inverted pendulum is decomposed into disjoint subsets. For some state of the inverted pendulum, one of these subsets is selected at random, constituting a pathway to a particular synapse of the neural network (an interconnection weight) when the state values are applied to the network. A particular weight becomes *eligible* if the signal passing through it causes its associated neuron to fire. Then the weight will be updated depending on the value of the reinforcement, $r(t)$.

This approach is a modification to a method previously defined by Michie and Chambers [39] where they similarly divided the state space into small parts but used a nonneural implementation. Anderson [40] also used this division scheme but employed error backpropagation to train a two-layer network.

Reinforcement learning will be described in greater detail when the generalization of the *Learning Automaton* is defined in Chapter 2.

Another successful demonstration of a neural network as a regulator is the one mentioned earlier by Ichikawa and Sawa [33].

1.4.1 Inverse Dynamics

Consider the problem where the output trajectory of a plant has been specified and it is desired to determine the control signal to the plant that achieves the desired plant output. If a unit could be designed that represents the inverse dynamics of the plant then, if the input to that unit is the desired plant output, then the unit will give the desired control signal. This is shown in Figure 8. This type of controller is a feedforward controller.

Figure 8 Inverse Dynamics Controller

Successful demonstrations of this approach have been shown by Psaltis, Sideris and Yamamura [42] and Kawato, et. al. [43]. Miller, et. al. [44] have demonstrated the use of

a Cerebral Model Articulated Controller (CMAC) network as a controller for robotic manipulators. CMAC is a lookup table technique for generating nonlinear functions.

Since the neural network is designed around one set of parameters, it is not robust with respect to changes in the plant parameters. The control signal that is generated is only due to the reference input, so it is tailored to the original plant dynamics.

1.4.3 Control by Identification of System Dynamics

Adaptive controllers seek to modify the control system structure to compensate for variations in plant dynamics. In order to do this, some method must be found to identify the changing plant dynamics and then use the identification to either modify the controller or select between various controllers which have been designed to meet system requirements based on selected parameter neighborhoods. Some examples of adaptive controllers include the Model Reference Adaptive Control (MRAC) and self-tuning regulators.

Narendra and Parthasarathy [46] have shown that parameter identification can be learned using recurrent configurations of dynamic neurons. Here the dynamic neurons are discrete time models instead of continuous time models. One identification scheme is shown in Figure 9. Note that TDL denotes a tapped delay line.

Before the neural network is installed in this configuration, it has been trained to model the input-output

function of the nonlinear plant. Then the configuration, a series-parallel configuration, gives an error which indicates the deviations in the nonlinear plant from the model. This error can then be used by a controller to determine what changes must be made to meet control specifications.

Figure 9 Identification of Nonlinear Plants

One application of this approach where neural networks have been successfully used was done by Karakasoglu, Sudharsanan and Sundareshan [15] and [55] in the control of robot manipulators with dynamic recurrent networks simulating digital filters. Muhsin, Sundareshan, Sudharsanan and Karakasoglu [45] and Sudharsanan, Muhsin and Sundareshan [56] have also demonstrated an application of this method for controlling nonlinear power generators. Backpropagation learning was used to train the networks in both cases.

1.4.4 Continuous Time Control

Most applications of neural networks to control system problems have been either with static networks or with dynamic networks operated in a static mode; i.e., the network is trained to operate in a discrete-time mode in a manner similar to a digital filter. This is due, in part, to the difficulty in training a dynamic, recurrent network to generate continuous-time waveforms. Of the training schemes described so far, the only ones possibly applicable to the design of a continuous-time controller would be the backpropagation schemes applied to trajectory learning in recurrent dynamic networks. Unsupervised learning is only applicable to static networks or to dynamic networks operated in a fixed-point mode as are most of the other methods previously described. The disadvantage of those backpropagation schemes is that the desired signal must be known *a priori* before learning can be attempted. In the case of simple linear systems, standard methods can be used to determine a control signal that meets some specified cost function; however, in the case of nonlinear plants which may not have a neat analytical representation, it may not be possible to analytically determine the desired control signal.

If it were only possible to use neural controllers when the desired control signal is known, there would be little advantage to neural controllers over more standard methods

of control. On the other hand, if it is possible for the neural network to learn a control signal for an arbitrary plant and cost function where the control law cannot be determined analytically, there would be a distinct advantage with using the neural network. A possible learning method will be described in Chapter 5 which points to an attractive solution to this problem.

1.5 Organization of Thesis

This thesis is organized as follows.

In Chapter 2, two methods of trajectory learning for dynamic, recurrent networks are described in detail.

The first method uses the concept of a learning automaton. This starts off as a trial and error method that learns by experience. If an action taken by the automaton is successful, that is if the action brings the output of the neural network closer to its objective, then that action will have a higher probability of being selected again. An action is defined as the modification of a parameter or a group of parameters in the neural network. As an adjunct to this method, the concept of incremental learning is defined, where the learning goals are generated which bring the neural network from some initial configuration to the final desired configuration in smaller steps. Incremental learning is used to lessen the problem of getting stuck at a local minima which is encountered in many methods of neural

network training.

The second method is a direct gradient calculation method similar to the method described by Toomarian and Barhen [18]. This method calculates the gradient directly by making small changes in a parameter and then calculating the change in the selected error criterion. This is a good approximation to the gradient when an accurate integration algorithm is used to calculate network dynamics.

In Chapter 3, the implementation of these two methods with regard to simulating the neural network in a learning environment is described. The specific algorithms and data structures are given along with details of the problems to be solved with the learning methods outlined in Chapter Two, viz., the implementation of the learning automaton and the generation of learning goals.

In Chapter 4, the simulation results are given for one benchmark trajectory, the circle trajectory. Several experiments are performed: some showing the effects of initial conditions on the ability to find a satisfactory solution to the problem, some showing the effects of using different specifications of the problem and finally, one showing the effects of a different type of learning automaton. The results of the simulations are given in plots showing the final outputs and also showing the progression of the learning. Some discussion is given concerning these results. A comparison is made between the learning automaton

approach to trajectory learning and the gradient approach. The data and final neural network parameters for the solutions to these problems are also given.

In Chapter 5, the learning automaton method is applied to two control problems -- a linear trajectory tracking problem and a nonlinear stabilization problem. The data and neural network parameters are given.

In the first case, a simple linear plant is defined for which a desired output is specified. The neural network is trained in a feedback configuration to generate a control signal which meets the output specification within a prespecified error tolerance. The final control signal and plant output are given along with network parameters and data showing the progression of the learning.

In the second case, it is desired to stabilize a nonlinear plant from some set of nonzero initial conditions. The same system configuration is used as for the linear case. In this case, it will be seen that incremental learning is not required.

In Chapter 6, final conclusions concerning the effectiveness of the learning automaton method with regard to control system design are given.

The major contributions of this thesis are the following:

1. Two new methods of neural network learning are proposed and their performance is evaluated. These

are the learning automaton approach and a new, simplified gradient descent method. These methods are described in detail along with theoretical justification for their use.

2. A method of determining control laws implicitly without the necessity of solving the system equations is discussed. This is an application of the learning automaton approach with the use of incremental learning applied to dynamic, recurrent neural networks.
3. A successful demonstration of trajectory learning and control learning with dynamic, recurrent neural networks is given.

Chapter Two

Learning Automata

2.0 Learning Automata

A significantly different approach in the training of neural networks might be to determine which parameters of the neural network, when incremented or decremented, reduce the error between the actual response and the desired response. A method such as this might be one in which parameters are selected based on some probability distribution. The probability of selection might be increased each time it can successfully be changed to reduce the error while the probability might be decreased if a change in the parameter increases the error. A method like this has been developed by Tsetlin [23] and enhanced by others. It makes use of a *learning automaton*.

In general, an automaton is a system whose states take on only certain fixed values, normally 0 or 1. The states, labeled ϕ_i , of an automaton evolve according to some preset rule based on the present states and any inputs.

A learning automaton has been defined by Narendra and Thathachar [22] by the following construction:

$$\text{automaton} \equiv (\mathbf{s}, \alpha, \beta, F, G)$$

where $\mathbf{s}$ = state $(\phi_1, \phi_2, \dots, \phi_n)$

α = output $(\alpha_1, \alpha_2, \dots, \alpha_r)$

β = input $(\beta_1, \beta_2, \dots, \beta_m)$

F = updating algorithm

G = output function $G : \mathbf{s} \rightarrow \alpha$

Note that boldface denotes a vector. The automaton interfaces to a structure called the *environment* (see Figure 10) which is a function mapping the automaton output to its input, or

$$\text{environment} \equiv E : \alpha \rightarrow \beta$$

The environment also serves to give feedback on the effectiveness of the learning through rewards and penalties. This will be discussed in greater detail below.

The structure of the learning automaton may take on several forms. One variation has to do with the structure; it can be either fixed or variable. The fixed structure automaton is not altered as learning progresses while the variable structure may have changes occur in either the state or output updating functions. Here structure refers to the way in which the learning automaton state or output is determined. Note that with a fixed structure automaton, there will be no actual learning.

Figure 10 Learning System

Another variation is in the state and output functions themselves which can either be deterministic or stochastic. A deterministic automaton has state and output functions which are deterministic; that is, a normal mapping function from the state to either the state or the output. The state and/or output functions for a stochastic automaton take the form of the probability of changing from one state to another based on the present state and the input.

Learning algorithms are generally in the form of *reinforcement schemes*. In a reinforcement scheme, the output of the environment is used to determine whether the action of the automaton has been favorable or unfavorable. The convention for a favorable response from the environment is $\beta_i = 0$, while an unfavorable response is denoted $\beta_i = 1$. When a favorable response is received from the environment, the structure of the automaton is altered to make that action occur more often. In the case of the deterministic

automaton, for the same inputs and state, the rule that defines how the states and outputs are updated is modified so that the favorable action is the one which occurs. In the case of the stochastic automaton, the probability of a favorable action is increased for a favorable response and reduced for an unfavorable one.

In the case of the stochastic automaton, the reinforcement scheme can be either linear or nonlinear. The most basic scheme is the *Linear Reward-Penalty* scheme where the state update probability function or output update probability function is changed by an amount proportional to the value of the probability. If a favorable response is received from the environment for action α_i then the probability for this action is increased through some reward function $g(P(\alpha))$ while the probabilities for all other actions are reduced by some linear function $h(P(\alpha))$.

$$P(\alpha_i) = P(\alpha_i) + g(P(\bar{\alpha})) \quad \text{for } \beta(\alpha_i) = 0 \quad (71)$$

$$P(\alpha_j) = P(\alpha_j) - h(P(\bar{\alpha})) \quad \text{for } j \neq i \quad (72)$$

Here, the notation, $\beta(\alpha_i)$, refers to the environmental feedback for the automaton output α_i ; $\beta() = 1$ for an unfavorable response, while $\beta() = 0$ for a favorable response. The overbar refers to the vector of possible actions.

In the case of an unfavorable response for α_i , the probabilities are changed according to the rules

$$P(\alpha_i) = P(\alpha_i) - h(P(\bar{\alpha})) \quad \text{for } \beta(\alpha_i) = 1 \quad (73)$$

$$P(\alpha_j) = P(\alpha_j) + g(P(\bar{\alpha})) \quad \text{for } i \neq j \quad (74)$$

Note that $g(P(\alpha))$ and $h(P(\alpha))$ are positive definite functions. Another reinforcement scheme, called the *Reward-Inaction* scheme, has $h(P(\alpha))$ equal to 0, so changes occur to the structure only for favorable responses from the environment. A modification of the Linear Reward-Penalty scheme is the Linear Reward- ϵ Penalty where the penalty function is modified by a positive coefficient $\epsilon < 1$.

Several researchers have shown successful applications of learning automata to the problem of telephone call routing. These include Narendra, Wright and Mason [24], and Narendra and Mars [25]. Meybodi and Lakshmivarahan [26] and Srikanthakumar [27] have successfully used learning automata in the study of waiting time in queueing systems. Hashim, Amir and Mars [28] have also successfully applied learning automata to the problem of image data compression.

2.1 Learning Automata and Neural Networks

The principal advantage of the learning automaton is its ability to determine optimal actions among a set of possible actions. This is discussed below under the topic of convergence. In the case of neural network training, it is desirable to determine which increments or decrements to parameter can be adjusted successfully to reduce some error

criterion and which cannot. In order to apply the learning automaton approach to neural network training, some set of actions must be defined which can be used to reduce the error criterion.

2.1.1 Learning Configuration

Define the set of possible actions to be the incrementing or decrementing of a neural network parameter by some fixed amount. The action of incrementing a parameter is distinct from decrementing a parameter. If a neural network has N neurons then the automaton will have $2*(N^2+2N)$ possible actions corresponding to the N^2 weights, N time constants and N nonlinearity gains. The learning automaton is stochastic; i.e., the actions are selected based on assigned probabilities.

A probability is initially assigned to each action. Since there is no *a priori* knowledge about a particular neural network as to which of its parameters can be altered to reduce some error criterion, the entropy with respect to learning is maximum at the beginning of training. Therefore a uniform distribution is used for the action probabilities at the beginning of training.

As learning progresses, the probability associated with each action is changed. This probability determines the relative frequency with which a particular action will be selected. The more successful an action is at reducing the error, the more likely it will be selected.

The environment for this configuration becomes the neural network itself and its associated error criterion. The feedback to the learning automaton is the success or failure of an action to reduce the error criterion. In the case of trajectory learning, the error criterion may be calculated with respect to the neural network outputs vis-a-vis the desired trajectory; in a control system problem, the error criterion is calculated with respect to the desired plant output or trajectory.

2.1.2 Learning Algorithm

A simple reinforcement scheme to use for the action probabilities is the linear reward penalty scheme. As will be discussed below, convergence of the learning automaton is not desirable. An example of a simple scheme to be used is

$$\begin{aligned} P(\alpha_i) &= P(\alpha_i) + 0.02P(\alpha_i) \quad \text{for } \beta(\alpha_i) = 0 \\ P(\alpha_j) &= P(\alpha_j) - 0.01P(\alpha_j) \quad j \neq i \end{aligned} \quad (75)$$

$$\begin{aligned} P(\alpha_i) &= P(\alpha_i) - 0.01P(\alpha_i) \quad \text{for } \beta(\alpha_i) = 1 \\ P(\alpha_j) &= P(\alpha_j) + 0.02P(\alpha_j) \quad j \neq i \end{aligned} \quad (76)$$

Referring back to equation (71) and equation (72), $g(P) = 0.02P(\alpha)$ and $h(P) = 0.01P(\alpha)$. These values are used in some of the experiments described in Chapter 4 and were selected on the basis of experimentation to give good results for learning.

If an action reduces the value of the error criterion, then the probability of selecting that action again is increased proportional to the value of the probability while

the values of the probabilities of the other actions are decreased. Conversely, if the action increases the value of the error, the probability of selecting that action is reduced, while the probabilities of the other actions are increased. In this manner, the probability of selecting actions can change as the sensitivity of the error to parameter changes changes.

Once it has been determined that an action has reduced the error, the corresponding change to the neural network parameter is retained. However, if the action increases the error, the corresponding parameter change is not kept. Therefore the only modifications to the neural network are those which reduce the value of the specified error criterion.

2.1.3 Convergence

Two types of convergence can be discussed with respect to the learning automaton approach to neural network training; convergence of the error and convergence of the automaton to some optimal action.

Convergence of the error is assured by the learning algorithm. For any set of initial conditions of the network and some error criterion, the initial error before learning starts will be some finite positive number, say E_0 . Because of the learning algorithm, changes to the neural network are only made if the error is reduced. Therefore a monotonic decreasing sequence of real numbers is generated. The error

is always a positive number, so the sequence is bounded. By the Monotone Convergence Theorem (see, for example, Bartle and Sherbet, [31]) the sequence converges. This is not to say that the error will converge to zero, but it will converge to some number.

Under certain conditions, the learning automaton will converge toward some optimal action depending on the type of reinforcement scheme used. Associated with each action, there is a probability of penalty, c_i . It has been shown (see Narendra and Thathachar, [22]) that if these probabilities are stationary, then the action probabilities will converge towards some optimal action. In the case of the linear reward-inaction scheme, convergence is assured.

However, this may not be desirable in the case of neural network training. The probability penalties are not known at the start of training; especially it is not known if their distribution is stationary. It is assumed that this is not the case, since the structure of the neural network is constantly changing during the training process. An action that may produce a favorable response at some point in the learning process may not provide a favorable response at some later time. For example, if a weight is continually decremented until it becomes negative, the response may be changed from something that is approaching the desired trajectory to something that may be different from what is desired. In addition, the neuron model only uses gains and

time constants which are nonnegative real numbers. The time constant is the decay time for the neuron. A negative time constant would force the neuron response to be increasing instead of decreasing for nonzero initial conditions, which is undesirable. A negative gain would switch the response of the nonlinearity which would have the effect of reversing a neuron's polarity. Therefore, convergence of the learning automaton to an optimal action is not desirable and will not occur when the reward-penalty reinforcement scheme is used since the probability of any action going to 1.0 is not possible with this scheme for a nonstationary environment.

2.2 Incremental Learning

One of the major problems associated with neural network learning is the tendency for the learning to progress for some time, with regards to reducing the value of an error criterion, but then reach a point where the error can no longer be reduced. In the case of trajectory learning, this occurs when the integral of the real error, $\zeta(t) - V(t)$, is zero but the value of error criterion, which is strictly positive, is not. If the integral of the real error is zero, then the gradient of the integrated error with respect to any of the network parameters must equal zero. This implies that the average error versus the system parameters is either at a maximum or a minimum. Assuming it is at a minimum, then any parameter change around the

neighborhood of the existing parameters can only increase the error.

This condition may be visualized by considering a surface in an N^2+2N+1 -space, where the N^2+2N axes correspond to the parameters of the neural network, and the final dimension corresponds to the error criterion. The error has been reduced vis-a-vis the parameters but has fallen into an energy well, called a *local minimum*, from which the type of parameter changes already used in the learning cannot remove it. Some radical change to the parameters is then required to enable learning to progress. It may be necessary to increase the error initially in order to enable learning to progress. It may also be the case that no change, other than restarting the learning from another set of initial conditions, may enable learning to progress past the error value achieved at the local minimum.

A solution to the problem of getting stuck at local minima is to control the evolution of the learning so that the integral of the real error is close to zero only when the strictly positive error criterion is close to zero. In this way, the problem of the learning progressing into a local minimum is minimized. This can be accomplished by setting small learning goals from the present state of the neural network to the final desired state. Such a strategy is particularly useful in the learning of continuous trajectories.

For example, assume that it is desired that a dynamic, recurrent neural network learn the trajectory

$$v(t) = 0.5\sin(\pi \cdot \text{time}), \quad \text{for } 0 \leq \text{time} \leq 1.0 \quad (77)$$

Let $v_0(t)$ be the initial trajectory of the neural network for some initial state of the neurons and some initial set of parameters. Note that the initial state must be nonzero; otherwise, the output of the neural network will remain at zero. Select an arbitrary number of learning targets, say 100. Then these learning targets are defined by

$$\zeta_n(t) = v_0(t) + \frac{n[\sin(\pi \cdot t) - v_0(t)]}{100}, \quad \text{for } \begin{cases} 0 \leq t \leq 1.0 \\ 0 \leq n < 100 \end{cases} \quad (78)$$

When $v_0(t) \approx \zeta_i(t)$ within some predetermined error bound, the next learning target becomes $\zeta_{i+1}(t)$. Learning progresses through these increments until the final desired target is reached. Figure 11 shows a succession of these desired trajectories which represent incremental targets, assuming that the initial trajectory of the network is close to zero.

Figure 11 Trajectory Targets

2.3 Neural Network Structure Variation

The actions of the learning automaton are already defined as incrementing or decrementing the parameters of the neural network. The size of these increments has yet to be determined. Since there are no criteria available for the selection of the increment size, it is set at some arbitrarily small value, then applied as a structure change to the selected parameter. If the error is reduced by the structure change, the increment is increased and tried again. This process is continued until the selected structure change is no longer successful at reducing the error. At that point, a new structure change is selected

based on the probability distribution in the learning automaton. This completely defines the actions of the learning automaton.

Once the error is within some specified bound, the next trajectory target is used for the calculation of the error criterion.

Chapter Three

Simulation

3.0 Simulation Structure

A program was written in C to simulate the neural network/learning automaton system. The major data structures and algorithms are described below.

Three sets of simulations are performed. The first two involve a comparison test between the learning automaton method of neural network training and a gradient descent approach. For comparison, one of the benchmark trajectories defined in the literature will be used. This is the circle trajectory. The third simulation set will be applications of neural networks to control problems.

The circle is generated by a neural network with two outputs; the first output generates a sine wave of set amplitude and frequency while the second output generates a cosine wave of the same amplitude and frequency, or

$$\begin{aligned} v_{o1}(t) &= A \sin(\omega_o \cdot t) \\ v_{o2}(t) &= A \cos(\omega_o \cdot t) \end{aligned} \quad (79)$$

The generation of the circle is shown in Figure 12.

Figure 12 Circle Generation

3.1 Data Structures

3.1.1 Neural Network Structures

The following data structures define the parameters and states of the neural network being trained. The dimension of the array is given by the value in brackets. Note that this is the convention for the C programming language.

1. $w_{ij}[t][f]$ - This two dimensional array holds the interconnection weights. It has size $t \times f$ where the first index is the

neuron to which the weight goes and the second index is the neuron the connection is coming from.

2. $\tau[t]$ - This array holds the relaxation time constants for the t neurons of the network.
3. $gain[t]$ - This array holds the nonlinearity gains for the t neurons of the system.
4. $v[t]$ - These are the neuron outputs.
5. $h[t]$ - These are the intermediate sums of the weighted outputs of the neurons at neuron t .

3.1.2 Learning Automaton Data Structures

The only learning automaton structure that needs to be stored is the discrete probability distribution function of the automaton actions. This is stored as an array of real numbers whose sum is 1 and has dimension $2(N^2 + 2N)$. Its use is described below.

3.1.3 Gradient Descent Data Structures

The gradient descent method uses the same neural network data structures as the learning automaton.

3.2 Algorithms

3.2.1 Learning Automaton Algorithms

A fourth order, Runge-Kutta algorithm is used to integrate the neuron differential equations for all simulations and plant differential equations for the control system simulations. A time increment of $0.02 \cdot T$ is selected

for the integration time constant, where T is approximately the period of the trajectory to be generated. This integration algorithm has a relative accuracy of 0.0001, which is accurate enough for the parameters and signals in question.

An output function, $\alpha(n)$, must be generated for the stochastic automaton structure. This output function maps the trial, n , into a selection of the appropriate action to take in a probabilistic fashion. Since these action probabilities are unknown at the start of the experiment, they are initialized to a uniform distribution. Then, as the experiment progresses and successful actions are found, a discrete probability density function is built up, with the probability for a particular action being increased or decreased by the reinforcement functions, equations (75) and (76).

As the density function is being generated it is used for parameter selection; i.e., action generation. This is done by generating uniformly distributed random numbers and then summing the numbers in the density function to create a distribution function until the generated random number is greater than the sum. The action is selected at the point where the sum of the densities is greater than the uniform random number. This is known as the inverse distribution method. The uniform random numbers are generated by the Lewis-Paine method [4].

A method of teacher forcing, similar to that used by Williams and Zipser [16] for trajectory learning, was used for the learning of the circle. The learning steps were used as output feedback to the neurons of the network instead of the actual outputs. When the fiftieth step was reached, teacher forcing was disabled, and the actual outputs were used until the training was complete. Teacher forcing was not used for the control learning problems.

In the case of trajectory learning, the teacher forcing signal is the desired trajectory learning increment function and is fed back to the other neurons in the recurrent configuration. In the case of control signal generation, however, the appropriate network output is not known, and therefore teacher forcing cannot be used, since the appropriate signal cannot be fed back into the recurrent configuration.

The actual trajectory learned for the circle is

$$\begin{aligned} v_{o1}(t) &= (1 - e^{-\alpha \cdot t}) \cdot \sin(\pi \cdot t) \\ v_{o2}(t) &= (1 - e^{-\alpha \cdot t}) \cdot \cos(\pi \cdot t) \end{aligned} \quad (80)$$

where α is used to control the trajectory rise time rather than try to force the network to generate the circle with an unknown rise time. The value of α was varied during the experiments to determine the consequences of varying the trajectory rise time. Note that the state of the neurons at the beginning of the trajectory is a set of random numbers

approximately equal to zero.

3.2.2 Gradient Descent Algorithms

For the most part, gradient descent algorithms have been used to train dynamic, recurrent neural networks to generate trajectories; for example, see Pearlmutter [17] and Toomarian and Barhen [18]. Therefore it is desirable to compare the learning automaton approach to trajectory learning with the gradient descent methods.

For calculation of the gradients, a simple approximation of the partial derivative is used:

$$\frac{\partial E}{\partial \rho} = \frac{E(\rho + \Delta) - E(\rho)}{\Delta} \quad (81)$$

In this case, $E(\rho)$ is the value of the error for the present value of ρ . The increment, Δ , is arbitrarily set to a small value -- 1.0×10^{-8} . By the definition of the derivative of a real function,

$$\frac{df(t)}{dt} = \lim_{\Delta t \rightarrow 0} \frac{f(t + \Delta t) - f(t)}{\Delta t} \quad (82)$$

as the value of Δt approaches 0, the derivative is more accurate. With a value for Δ of 1.0×10^{-8} , the gradient will be very accurate, relative to the values of the trajectories being generated.

This method of gradient calculation was selected over others; such as Toomarian and Barhen [18], and Pearlmutter [17], because of its simplicity. It was also found to be more accurate than the other methods because of the accuracy

of the integration algorithm used. In the case of these other gradient methods, only a forward Euler method was used, which is somewhat inaccurate, on the order of 0.1, or about one significant digit. The inaccuracy causes errors to build up in the calculations which in turn will cause errors in the final gradients.

As a modification to normal gradient descent, the effectiveness of the gradient was tested before any permanent change was made to the neural network structure. In some cases, the neighborhood of the gradient around the parameter being tested may be very small, and when the gradient is multiplied by the learning rate, the resulting change in the parameter may cause the error to increase instead of to decrease. Therefore, the change in the parameter was tested before a permanent change to the network was made. If the error was increased, smaller values of the learning rate were tried until a resulting change in the parameter did indeed reduce the error or, after three unsuccessful attempts, the parameter was not changed.

The parameters were selected for testing in an arbitrary order, and a parameter was not tested again until all of the other parameters had been tested. This constitutes one *loop*.

A method of teacher forcing, similar to the one used by Toomarian and Barhen was used. In their scheme, two feedback signals are generated; however, only one signal is input to

one output neuron. For the circle trajectory,

$$\begin{aligned} \text{if } v_{out_{des}}(t) &= k \cdot \sin(\pi \cdot t) \\ \text{then } v_{TF} &= \lambda \cdot [k \cdot \sin(\pi \cdot t)]^{\frac{2}{9}} [k \cdot \sin(\pi \cdot t) - v_{out}(t)]^{\frac{7}{9}} \end{aligned} \quad (83)$$

and

$$\begin{aligned} \text{if } v_{out_{des}}(t) &= k \cdot \cos(\pi \cdot t) \\ \text{then } v_{TF} &= \lambda \cdot [k \cdot \cos(\pi \cdot t)]^{\frac{2}{9}} [k \cdot \cos(\pi \cdot t) - v_{out}(t)]^{\frac{7}{9}} \end{aligned} \quad (84)$$

where $v_{out_{des}}(t)$ is the desired output and $v_{TF}(t)$ is the teacher forcing signal.

Toomarian and Barhen use

$$\lambda(\tau) = 1 - e^{-E(\tau)} \quad (85)$$

however this was found to be difficult to initialize. In this case, τ , is the loop count. When the simulation is started, there is an initial value for the error. When this is substituted into equation (82), λ instantaneously changes and the teacher forcing instantaneously changes. This changes the error and an unstable situation occurs. If an initial value of error can be selected such that this instability does not occur, then equation (82) is satisfactory.

A better expression for λ was found to be

$$\lambda = \begin{cases} 1.0, & \text{for error} > 0.9 \\ 1.0 - 0.01 \cdot (\alpha), & \text{for } \alpha > \text{loop count at error} = 0.9 \end{cases} \quad (86)$$

This was found to have better convergence properties than the coefficient used by Toomarian and Barhen.

3.3 Program Structure

The program structure is essentially modular and linear. The program starts out with initialization of the parameters before learning commences. Integration and random number generation routines are in separate modules. The program proceeds to reduce the errors until the final objective is achieved or until no further progress in learning can be achieved. Final results are written to a data file.

3.4 Comparison of Methods

In order to determine the functionality of the learning automaton method of training dynamic recurrent neural networks, some method of comparison must be made between the automaton method and the several gradient descent methods. The following table summarizes the comparison of memory and program requirements for the learning automaton approach and the three gradient descent approaches discussed above.

As can be seen, the simple gradient method has better properties than the other gradient methods for the same set of initial conditions and provides a better comparison for the learning automaton method. In the bottom row of the table, a loop for the gradient descent method has been previously defined as one gradient calculation for each parameter. A loop for the learning automaton method is defined to be an action, so this comparison for the methods

Table I Comparison of Learning Methods

	learning automaton	Pearlmutter	Toomarian -Barhen	simple gradient
computational complexity	medium	medium	high	low
memory	$O(N^2)$	$O(N^3)$	$O(N^3)$	$O(N^2)$
accuracy	high	low	medium	medium
computational time	2000 loops	10000 loops	200 loops	200 loops

may not be appropriate.

Chapter Four

Trajectory Learning Results

4.0 Simulation Environment

The simulations were run on a personal computer compatible with an IBM PC/AT using an Intel 80386 processor with an 80387 coprocessor, both operating at 25 MHz.

For each trajectory learning simulation, a dynamic recurrent network was used consisting of six neurons. Four neurons were hidden nodes and two neurons were used as output nodes. There were no input nodes.

4.1 Circle Trajectory

The results of three runs for learning a circle trajectory using the learning automaton approach are shown in the experiments below. The tables list three sets of network parameters while the figures show the actual trajectories output by the neural network. The time to complete the simulations ranged from 5 hours to 24 hours.

The results shown are only for the initial conditions used; that is, for the initial states of the outputs of the

neurons. If a trajectory is started from a different set of initial conditions, the desired trajectory will not be achieved.

4.1.1 Experiment 1

For the first experiment, the weights were initialized to 0, the time constants were initialized to random numbers centered around 6.0 and the gains were initialized to 10.0. The initial neuron outputs were initialized to small random numbers centered around zero. The results of the simulation are shown below.

Figure 13 shows the parameter changes or actions that were attempted by the automaton for each learning increment. This shows how the automaton gained experience with the environment as the learning progressed. For this experiment, teacher forcing was used for the first fifty learning steps and was then removed. Note that learning appears to be easier when teacher forcing is active. However, teacher forcing only teaches the network how to output the desired trajectory if that trajectory were already present. Therefore at step 50, when the teacher forcing is removed, the network output is not the desired output at step 50; it is something similar but with a much lower amplitude.

After step 50, learning becomes more difficult, as the network must now meet the learning goals on its own. Figure 13 shows that learning increased in difficulty until step 82, then the automaton had enough experience to begin making

better selections. This is not always the case, as later experiments show.

Figure 13 Actions per Learning Increment - Experiment 1

Figure 14 shows the final trajectory after learning has been completed. The time over which trajectory learning was simulated was 4.0 seconds. In this case, the simulation time was extended to 8.0 seconds to see how stable the circle trajectory was. It can be seen that the trajectory is approaching a limit after 8.0 seconds, so the trajectory is stable.

Figure 14 Final Trajectory - Experiment 1

The final neural network parameters are in the tables below.

Table II Neuron Interconnection Weights

from/to	0	1	2	3	4	5
0	-0.008	0.22325	-1.5005	7.4225	-0.01	-0.05175
1	0.008	0.03775	-0.02475	5.8135	0.00025	0.00025
2	0.0105	0.0545	0.00075	2.54275	-0.00025	0.00025
3	0.00025	-0.00075	-0.00975	0.05225	0.0	0.0
4	0.0975	0.0665	1.0395	7.09625	0.1	-0.00075
5	-0.214	3.0665	-5.73675	-13.1578	0.14475	-0.00125

Table III Neuron Time Constants and Gains

neuron	tau	gain
0	3.982919	11.47875
1	1.66565	10.69375
2	2.402844	13.7225
3	1.932819	7.62575
4	6.916434	11.56275
5	2.401347	10.41225

4.1.2 Experiment 2

It is desired to determine what the effect of increasing the learning time might have on the learning performance. For this experiment, the learning time was increased from 4.0 seconds to 5.0 seconds with a proportional increase in the error tolerance from 0.06 to 0.075 to account for the increased time of integration of the error criterion. In addition, the rise time is reduced from 1.2 seconds to 0.8 seconds.

The parameters were initialized randomly to the same range of values as for the previous experiment.

Teacher forcing was used only for the first 25 steps instead of the first 50 steps. Since teacher forcing is only used to get the learning started more quickly, it is desired to determine if it can be taken off more quickly so that real learning may start earlier.

These conditions give similar results as those of the previous experiment as is seen from the trajectory plot in Figure 15.

Figure 15 Final Trajectory - Experiment 4

A plot for actions per learning increment is shown in Figure 16. Note that learning became more difficult after step 85. This may be due to several factors; the inability of the neural network to generate a trajectory that differs from the desired trajectory or, more possibly, the reduction in the ability of actions to reduce the error; that is, where a particular action may have reduced the error by a certain amount δ , the action may later only reduce the error by only a fraction of δ . In this case more actions need to be taken to meet the learning target.

The neural network has been successful in learning the

desired trajectory with this set of operating conditions.

Figure 16 Changes per Learning Step - Experiment 4

Table IV Neuron Interconnection Weights - Experiment 2

from/to	0	1	2	3	4	5
0	0.021725	-0.00048	-0.35855	-0.0038	0.0	0.0
1	-0.0748	0.000050	1.47298	0.16038	0.0	0.0
2	-0.000275	-0.0001	0.141325	-0.050	0.0	0.0
3	0.020475	0.061	-0.165525	-0.00988	0.0	0.0
4	-36.44003	-0.291	140.679	-6.30725	0.10038	-0.02148
5	-21.15035	11.12045	-127.9962	23.8286	0.03058	0.061525

Table V Time Constants and Gains - Experiment
2

neuron	tau	gain
0	3.313519	10.916125
1	4.484987	9.686070
2	0.957068	9.236622
3	46.049375	8.068208
4	11.283719	10.165453
5	5.45620	15.918817

4.1.3 Experiment 3 - A Nonlinear Reinforcement Scheme

It is desired to continue the trend investigated in the previous experiment; that is, reduce the trajectory rise time and determine its effect on the ability of the neural network to accurately generate the desired trajectory. However, it was found that the learning automaton actions were not adequate for training the network for this trajectory. Some of this failing is due to the learning automaton only modifying one parameter at a time; there is no provision for determining the effects of modifying more than one parameter. Since the neural network is nonlinear, the effect of changing more than one parameter at a time will not be the same as the combined effect of changing them individually.

For this experiment, a different learning method was used with somewhat different initial conditions. A nonlinear reinforcement scheme is used. It is nonlinear in the sense

that the probability of selection of a particular action is changed in a nonlinear fashion instead of a linear method as given in equations (75) and (76).

In addition, a new set of learning automaton actions is created, where multiple parameter actions are allowed. The number of actions now increases. Consider a six neuron network. There are 36 weights, 6 time constants and 6 gains. The number of possible actions is now the number of permutations of the 36 weights multiplied by the number of combinations of time constants and gains. The number of permutations of the weights taken from 1 at a time to 6 at a time needs to be used since weight w_{ij} is not the same as weight w_{ji} or $36!$, while the number of combinations of 6 things taken from 1 to six at a time is needed for the time constants and gains. The number of actions in this case is

$$\text{number of actions} = (N^2 + 2 \cdot N)! \quad . \quad (87)$$

In order to reduce this number to something more manageable, the actions are limited to those for which the number of changed parameters is less than or equal to 10. However, even with this reduction, the number of actions is still quite large. In addition the probability density function becomes 10-dimensional.

As a result of the above issues, the reinforcement scheme is changed to take into account this increase in complexity. The successful actions are stored in a

repertoire. An action is added to the repertoire if it can be used successfully to reduce the error. If an action in the repertoire does not successfully reduce the error, it is removed. Once all behavior modifications in the repertoire have been used, new behavior modifications are selected randomly, based on a uniform distribution of all possible behavior modifications. If it takes fewer behavior modifications to reduce the error to its target value than are in the repertoire, then those behaviors are not used for achieving the learning target. When rewarded, the probability for an action in the repertoire stays at its previous value, whereas for a successful action not in the repertoire it is increased from its value in the uniform distribution to some other value. When penalized, the action probability is reduced to its value in the uniform distribution. In addition, actions are removed from the repertoire if they are not used; that is, if the behaviors ahead of it in the repertoire are sufficient to meet some number of learning goals, say 10, then the unused behavior is removed from the repertoire to make room for others that might be needed.

The learning algorithm is described in the following steps:

1. At the start of learning, parameters are selected randomly, except gains which are set to 10.0 on hidden layer neurons and 1.0 for output layer

neurons. These values are not changed during learning.

2. If an action has been used successfully, it is saved in a repertoire of learning actions.
3. Learning proceeds stepwise incrementally as before.
4. In any learning step, the learning actions in the repertoire are used first. Once these have been exhausted, new actions are selected at random until the step has been successfully learned.
5. Successful learning actions are placed at the bottom of the repertoire. Unsuccessful actions are deleted.
6. The size of the incremental changes to the parameters are scheduled based on the learning step and the size of the error; furthermore, these are scheduled to decrease with the step and the error.
7. The duration of the learning is started out with a time of 2.0 seconds. It is increased incrementally after the 50th step until it reaches 4.0 seconds, or two complete cycles of the circle. This was done to reduce computation time.

The results using this method were better than with the linear method used in the first two experiments, as is shown in the figure below. These results compare with the same results achieved by Toomarian and Barhen [16] for the case where learning was started with the values of the output neurons on the circle. This level of accuracy could not be

achieved with their method when learning was started at approximately the same initial conditions as in this experiment.

One reason for the better results in this case is that the trajectory rise time was decreased to 0.2 seconds, so the waveform essentially stopped increasing in amplitude before the end of the simulation time during training. Also, since more than one parameter can be changed for each behavior modification, other nonlinear effects come into play which cannot occur with the linear reinforcement scheme of the previous experiments.

In this case, learning was restricted to changes in the weights and time constants. Other experiments would have to be done in order to determine what effects would occur if learning was extended to the gains. This would have the disadvantage of increasing the amount of memory required to store the behavior modifications.

Figure 17 Trajectory From Nonlinear Learning

Figure 18 shows the number of behavior modifications stored per learning step. In this case, learning is easier at the beginning, becomes difficult, but then becomes easier again.

Figure 18 Behavior Modifications Stored

Table VI Neuron Interconnection Weights - Experiment 3

from/to	0	1	2	3	4	5
0	-0.02847	0.37719	0.41382	0.89337	-0.38815	-0.16833
1	-0.44113	-0.37424	-0.12475	0.35261	-0.15201	-0.33707
2	-0.10397	-0.41506	0.1331	-0.155223	0.57979	0.071725
3	-0.02099	0.06476	0.069281	0.126604	0.033235	-0.18256
4	0.35916	-0.3247	0.24363	0.56533	0.33486	-3.10767
5	0.1637	-0.25548	0.080121	0.05760	0.46822	0.40242

Table VII Time Constants and Gains ~
Experiment 3

neuron	tau	gain
0	4.54595	10.0
1	1.387306	10.0
2	56.813994	10.0
3	5.472248	10.0
4	2.419643	1.0
5	6.179582	1.0

4.2 Experiment 4 - Gradient Descent Learning

For this experiment, the gradient descent algorithm was used. Initial conditions used were the same as used by Toomarian and Barhen [18]; that is, the weights were set to positive and negative random numbers of magnitude 0.1 centered around 0, the time constants and gains were set to similar conditions as before, and the initial neuron outputs were set to random numbers of magnitude 0.1 centered around zero.

The error reduction versus time is shown in Figure 19. The error is approximately the same value as was achieved by Toomarian and Barhen [16] in a fewer number of loops, and the computation time is less. Their error graph is shown in Figure 20.

The final output is shown in Figure 21. Although, a circle is not achieved, there is a stable limit cycle with an amplitude of about half of the desired 0.5. Learning progressed until the error could not be reduced past this point. This also compares favorably with the Toomarian and Barhen approach.

Figure 19 Error Vs. Time for Gradient Learning

Figure 20 Toomarian & Barhen Error Graph

Figure 21 Final Trajectory from Gradient Learning

The main purpose of this experiment is to compare the gradient descent method with the learning automaton approach to trajectory learning. In terms of computation time, the two methods are equivalent, both reaching solutions in about 5 hours. In terms of the accuracy of the results for similar sets of initial conditions, the learning automaton approach is shown to be superior. However, its real utility will be demonstrated in the design of controllers.

Chapter Five

Neural Control

5.0 Control System Design

It is desired to use a neural network as a controller in a feedback configuration. However it is desired that the neural network be more than just a compensator; that is, it is not used in a configuration where its input is an error signal. Instead, the inputs to the network are the reference input and plant feedback -- either output or state --, and its output is the control signal(s) to the plant. This is shown schematically in Figure 22.

A configuration such as this has been shown to be effective in control systems applications by Ichikawa and Sawa [33]. However in their application, static neurons were employed in a perceptron type of arrangement and a genetic learning algorithm was used for training.

In this chapter, the learning automaton approach will be used for training with dynamic neurons in the network. The learning steps will now be generated from the desired

Figure 22 Control System Configuration

output of the plant instead of the desired output of the neural network.

Note that in all cases, the term *input node* is used to identify input signals to the neural network and does not imply that the input node is a neuron.

5.1 Control of a Linear Plant

It is desired to generate a control signal for a plant having a transfer function

$$G(s) = \frac{1.0}{s(s+1.0)} \quad . \quad (88)$$

The desired output for the plant is

$$y(t) = \begin{cases} 0.25 * \text{time}, & \text{for } 0 \leq \text{time} < 2.0 \\ 0.5, & \text{for } \text{time} \geq 2.0 \end{cases} \quad . \quad (89)$$

The learning automaton approach with linear

reinforcement and one parameter actions is used for this experiment. The error criterion to be used by the environment is now relative to the output of the plant instead of the output of the neural network. Incremental learning is also used with 50 learning steps.

The neural network will consist of two input nodes, five hidden nodes and one output node. The inputs to the network consist of the desired output, shown in equation (89), which will act as the reference input and the actual output of the plant for feedback. The output neuron is connected directly to the plant. This is shown schematically in the figure below. Note that the network is a recurrent one so that each neuron is connected to all other neurons. For clarity, not all of the connections are shown in the diagram.

This particular network was selected for the experiment because it was the same size as the configuration used to generate the circle trajectory. It was assumed that the same type of signal would be required from the output, in that it would have a similar spectrum and amplitude, although not necessarily exactly the same ones.

The final system parameters and signals generated are shown below.

Figure 23 Neural Network/Plant Configuration

Table VIII Neuron Interconnection Weights - Control System

to/from	0	1	2	3	4	5
0	8.17739	-0.01354	-0.01909	-0.04654	-2.42612	0.00228
1	-9.6699	0.03896	0.01682	0.03016	-4.9379	-0.03436
2	-4.94147	0.006823	0.06394	-0.08458	4.91281	0.037213
3	-4.17068	0.03779	0.06410	0.000921	-7.33842	-0.02187
4	7.30146	-0.00828	-0.00586	-0.02034	19.5449	0.00184
5	12.20746	-0.02826	-0.34186	0.03763	2.4321	0.05052
6	43.0403	-0.00347	0.00339	-0.01193	42.9410	0.04573
7	13.2360	-0.00764	-0.03016	0.0250	9.76821	-0.05278

Table IX Time Constants and Gains

neuron	time constant	gain
0	25.0	50.0
1	25.0	8.241445
2	1.706567	9.353442
3	25.0	42.847826
4	25.0	50.0
5	25.0	16.477181

Figure 24 Generated Control Signal

Figure 25 Desired and Actual Outputs

Figure 26 Output Error

In this case, the neural network has met its performance criterion. Note that in Table IX, several of the time constants and gains were attained an upper limit during the learning; 25.0 for the time constants and 50.0 for the gains. This was done to determine the effect of limiting of these values as the learning progressed. In this case, the neural network was still able to learn to generate a control signal to meet the specifications. As shown in Figure 26, the error was kept below 0.05 for most of the trajectory. There are no instabilities present in the control signal.

The low magnitude of many of the weights suggests that fewer neurons are needed than were used in the network configuration. Most of the error is at the points where the signal changes the most; that is, at the beginning and at the point of inflection at 2.0 seconds. Clamping the time constants at 25.0 shows that most of the learning effort went into meeting the high frequency requirements of the trajectory.

Even though the plant is linear, note that the control signal does not appear as an output of a linear controller might; a smoother waveform would be expected. But this control still meets the requirements.

The robustness of the neural controller to changes in the initial conditions must be determined. Let the initial outputs of the neural network and the initial states of the plant be set to random numbers centered around zero with a

maximum magnitude of 0.1. The results for this simulation are shown in Figure 27. As can be seen, the output of the plant still tracks well for this change, so the neural controller is indeed robust.

Figure 27 Response for Random Initialization

5.2 Nonlinear Plant Stabilization and Control

5.2.1 Stabilization

As mentioned in Chapter 1, Ichikawa and Sawa have applied neural networks to the problem of stabilizing a nonlinear system. They used a perceptron-like network of layers of static neurons to achieve their goal. For comparison, a dynamic recurrent network is used to stabilize the same system. The learning automaton will again use the single parameter actions with linear reinforcement.

The system dynamics are given by

$$\frac{d^2x}{dt^2} = \text{sgn}(x) \cdot u_{in} \quad , \quad (90)$$

where x is the plant output and u_{in} is the plant input.

The error criterion to be used for this experiment is to simply minimize the difference between the plant output at some specified time, in this case 10 seconds, and zero. So incremental learning as used in the previous experiments is not needed.

Obviously, if the plant is at zero initial conditions, then it will remain at zero, so some set of initial conditions is specified. In this case, the output of the plant can either start at +1 or -1. The derivative is set equal to 0 at $t=0$.

It then becomes the task of the learning automaton to train the network to force the system to zero for both sets of initial conditions. State feedback will be used for the system, where the states are defined to be the position variable, x , and its time derivative, dx/dt . The error criterion to use is then to drive x and dx/dt to zero for both sets of initial conditions.

The configuration of the system together with the neural network controller is the same as that shown in Figure 22, with the exceptions that there is no reference input, the neural network has eight neurons instead of six, where seven neurons comprise the hidden nodes and one neuron is the output node, and a gain of 10.0 has been placed

between the network and the plant. The inputs to the network are the states of the plant. The configuration was selected after several other network sizes were tried unsuccessfully.

Because the plant is characterized by a double integration to get the position response, longer periods of simulation are required to achieve the desired response, on the order of 20 to 40 seconds. It was found that the longer periods of integration caused the neuron outputs to become very large due to the presence of the time constant term on the right hand side, so the neuron equation used is

$$\frac{dV_i}{dt} + \tau_i V_i = \tanh(g_i \sum_j w_{ij} V_j) \quad . \quad (91)$$

Note that the τ_i factor has been deleted from the right hand side of the equation. The loss of this time constant term slows down the response and reduces the magnitude of the neuron outputs, so a gain of 10.0 is placed between the neural network and the plant. This factor of 10.0 in the configuration speeds up the response.

Figure 28 and Figure 29 show the evolution of the state errors versus training loop. Note that these are the errors at the end of the trajectory training time, not the initial values. An initial training strategy was to allow only changes which reduced the errors in both states. It was found, however, that the training might reach a point where no further reduction in the error was possible. An alternate

strategy which allowed small increases in one error as long as the other error decreased was more successful. As can be seen in Figure 28, there was a small increase in the error in dx/dt around step 200. This allowed further decreases in the other error. Note that the error curves evolve in a manner similar as those seen for gradient learning.

Figure 28 Error in dx/dt Versus Training Loop

Figure 29 Error in x Versus Training Loop

Expediency is defined (see Narendra and Thathachar [22]) as the frequency of receiving a penalty reinforcement, $\beta = 1$, from the environment. As shown in Figure 30, after about step 300, the expediency is monotonically increasing. This indicates that, for this particular example, no further learning is possible; that is, the errors can no longer be reduced.

Figure 30 Expediency

The following figures show the final control signals generated by the neural network for each set of initial conditions and the response of the plant.

Figure 31 Control Signal For $x(0) = +1$

Figure 32 Plant Output For $x(0) = +1$

Figure 33 Control Signal For $x(0) = -1$

Figure 34 Plant Output For $x(0) = -1$

So, the learning automaton method can be used to train a dynamic recurrent network to stabilize a nonlinear plant.

The final weights and time constants for the network

are shown in the tables below. The weights were initialized to small random values centered around zero with a maximum amplitude of 0.05. The time constants were initialized to 1.0, since the time constant of the plant has the same order of magnitude, and the gains were initialized to 2.0 for the same reason. Note that for a time varying input, the neuron response is much faster for a large nonlinear gain.

Table X Neuron Interconnection Weights ... first half

to/from	0	1	2	3
0	0.016551	-0.016864	-0.047622	-0.008858
1	1.286964	0.013887	-0.013694	-0.023658
2	-0.047948	0.044046	0.047275	-0.798193
3	0.032955	0.040068	0.035103	0.018619
4	-0.048456	0.048193	0.038429	-0.01450
5	-0.001784	0.009158	0.018097	-0.110562
6	0.032033	-0.001841	-0.010608	0.03710
7	0.007909	-0.013052	0.049913	-0.198926
8	0.032124	-0.009596	-0.047206	0.059478
9	-0.001522	-0.043383	0.039006	0.185164

Note that since there were fewer loops attempted during the learning in this experiment, about 300, versus previous experiments, on the order of several thousand, there is not much change in the parameters from their initial values, especially the time constants and gains. This problem appears to be a much simpler problem to solve for the neural network/learning automaton than the previous trajectory and

Table XI Neuron Interconnection Weights ... second half

to/from	4	5	6	7
0	0.023814	0.015311	0.011125	-0.048974
1	-0.036690	-0.049258	-0.042540	-0.042793
2	-0.012306	-0.024522	-0.005366	-0.038861
3	-0.006418	0.009793	-0.003575	0.004720
4	0.011623	0.023152	-0.054862	-0.044257
5	-0.043197	0.028011	-0.043582	0.033647
6	0.013097	0.018003	-0.038408	0.036652
7	0.017717	0.103699	0.068462	-0.009631
8	0.079204	0.001854	0.016817	0.04799
9	0.018967	-0.367236	0.029181	-0.047887

control system experiments.

In this case, it is difficult to make a comparison between the learning automaton approach and the genetic learning approach used by Ichikawa and Sawa. They used a population size of 20 with a maximum number of generations of 100. Ten loops were run per generation. This gives a total loop number of 20000, which is much larger than the number of loops run with the learning automaton approach, which ran about 200 loops before the error ceased to be reduced, possibly due to a local minimum. Ichikawa and Sawa used static neurons operating with zero-order holds in order to mimic the operation of a digital filter, whereas a continuous-time simulation was run with the learning automaton and the dynamic neurons. After training was complete, stabilization also occurred in roughly the same

amount of time, between 10 and 20 seconds. In terms of loop count and the number of individual neural networks trained, the learning automaton approach has an advantage.

Table XII Time Constants & Gains

neuron	tau	gain
0	1.0	2.0
1	1.0	2.012
2	1.0	2.0
3	1.0	2.0
4	1.0	2.0
5	1.0	2.45360
6	1.0	2.0
7	1.0	2.0

Although not specifically performed by Ichikawa and Sawa, it is desirable to know if the system stabilizes for other sets of initial conditions. The results for another set of initial conditions are shown in Figure 35.

As can be seen from the graph below, the neural network has been able to generate control signals which stabilize the nonlinear plant with a response that is similar for the case where $x(0) = \pm 1.0$. The plant response appears to be scaled down by 0.5; which is same scaling as the size of the initial conditions.

Figure 35 Output for Alternate Initial Conditions

For the stabilization experiment, the neural network was trained to perform two tasks; that is, stabilize the plant for two sets of initial conditions. This is in contrast to the linear tracking problem where the network was only required to solve one problem. This suggests that adaptability of the controller may be a result of the number of cases that the network is trained to handle.

5.2.2 Nonlinear Control

Consider the nonlinear system defined by the following state equations:

$$\frac{dx_2(t)}{dt} = u(t) - x_1(t) \cdot x_2(t) \quad , \quad (92)$$

and

$$\frac{dx_1(t)}{dt} = x_2(t) \quad . \quad (93)$$

It is desired to design a tracking controller for this system to meet the requirements defined in equation (89). The single parameter action automaton with linear reinforcement is used for training, along with incremental learning. For this problem, 50 learning increments are used with a network of eight neurons, comprising seven hidden nodes and one output node. The error criterion used is the integral of the absolute value of the error with a target final value of 0.06. This network size was arrived at by experimentation.

As can be seen in Figure 36, the neural network has learned to achieve its results quite well. The control signal generated by the neural network and the final parameter values are shown below, along with a plot of the error. These results are comparable to the results achieved with the linear plant.

Figure 36 Neural Network Control Signal

Figure 37 Desired and Actual Plant Response

Figure 38 Error For Nonlinear PLant

Table XIII Neuron Interconnection Weights 1 - Nonlinear Control System

to/from	0	1	2	3
0	-0.0083	0.06121	0.05821	-0.2779
1	0.03158	-0.00653	0.03078	0.04703
2	0.04721	0.02966	0.02840	-0.00585
3	0.03681	0.01039	0.00174	0.01648
4	-0.01119	0.00793	0.04702	0.01321
5	-0.02525	-0.02407	0.05366	0.04230
6	-0.02693	-0.08295	0.02417	-0.03866
7	0.04036	0.01727	-0.04265	-0.00219
8	0.03933	-0.07188	0.02071	-0.02021
9	-0.03451	0.07220	-0.02062	-0.0010

Table XIV Neuron Interconnection Weights 2 - Nonlinear Plant

to/from	4	5	6	7
0	-2.6299	0.02762	0.01421	0.03657
1	3.03528	-0.00066	0.00221	-0.02047
2	4.74659	0.01068	0.04052	0.0310
3	8.62614	-0.04197	-0.02234	-0.04607
4	3.79768	0.02522	-0.01270	-0.0219
5	2.1440	0.04487	-0.04050	0.023643
6	-2.06235	-0.02081	0.05969	0.05319
7	-1.0304	-0.07284	0.00602	-0.01096
8	-43.4050	0.03050	0.01070	0.03929
9	7.2743	0.02079	-0.03767	-0.01245

Table XV Time Constants and Gains

neuron	tau	gain
0	6.905936	9.455895
1	4.412911	9.411852
2	3.895649	8.078579
3	30.0	50.0
4	30.0	49.9999
5	2.533961	11.550721
6	7.385071	10.499654
7	30.0	11.425292

As with the neural control of the linear plant, it is desired to determine if the controller is robust with respect to alternate initial conditions. The figure below shows that this is again the case. In this instance, the same type of initial conditions are used; that is, random

numbers centered around zero with a maximum magnitude of 0.1. So this neural controller is robust with respect to changes in the initial conditions.

Figure 39 Response for Random Initialization

Chapter Six

Conclusions

6.0 Contributions

It has been shown that the method of the learning automaton can be successfully applied to the training of dynamic recurrent neural networks. This method is a formalization of the method of reinforcement training that has been used successfully with static networks in the past. The learning automaton method enables the neural network to gain experience with its environment and be trained based on that experience.

It has been demonstrated that dynamic, recurrent neural networks can be trained to generate the circle trajectory, learn to generate control signals to linear plants based on a specification of the desired output of the plant and stabilize a nonlinear plant. This training was accomplished with the learning automaton. These controllers have been shown to be robust with respect to changes in the network and plant initialization.

In addition, a simplified gradient descent algorithm

has been described for training dynamic, recurrent neural networks to generate trajectories. This method compares favorably with other methods. It has been shown that training neural networks with the learning automaton has accuracy and flexibility advantages over the gradient descent methods.

6.2 Topics for Further Investigation

Many of the parameters used in the learning automaton approach have been selected based on trial and error. Some investigation will have to be performed in the future to determine optimum values for such parameters as:

1. Increment sizes for changes to the network parameters,
2. Optimum reinforcement functions for rewards and penalties,
3. Less memory intensive methods for constructing the learning automaton to handle more complicated learning actions; i.e., changes to the neural network involving more than one parameter,
4. Defining more optimal training targets for the incremental learning
5. Most importantly, how to determine an optimum size for the neural network vis-a-vis a particular problem to be solved.

In addition, the learning automaton method must be applied to train neural networks to control more complex,

nonlinear systems.

References

1. W. McCulloch. and W. Pitts, "A Logical Calculus of Ideas Immanent in Nervous Activity", *Bulletin of Mathematical Biophysics* 5, pp. 115-133, 1943.
2. F. Rosenblatt, *Principles of Neurodynamics*, New York: Spartan, 1962.
3. J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities", *Proceedings of the National Academy of Sciences*, 1982, pp. 2554-2558.
4. T. Lewis and W.H. Payne, "Generalized Feedback Shift Register Pseudorandom Number Algorithm", *Journal of the Association for Computing Machinery*, Vol. 20, No. 3, pp. 456-468, July, 1973.
5. P. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Ph. D. Thesis, Harvard University, 1974.
6. D. Parker, "Optimal Algorithms for Adaptive Networks: Second Order Back Propagation, Second Order Direct Propagation, and Second Order Hebbian Learning", presented at IEEE First International Conference on Neural Networks, San Diego, 1987.
7. D. Rumelhart, G.E. Hinton and R.J. Williams, "Learning Representations by Back-Propagating Errors", *Nature* 323, pp. 533-536, 1986.
8. F. Pineda, "Generalization of Back-Propagation to Recurrent Neural Networks", *Physical Review Letters* 59, pp. 2229-2232, 1987.
9. L. Almeida, "A Learning Rule for Asynchronous Perceptrons with Feedback in a Combinatorial Environment", presented at IEEE First International Conference on Neural Networks, San Diego, 1987.
10. R. Rowher and B. Forrest, "Training Time Dependence in Neural Networks", presented at IEEE First International Conference on Neural Networks, San Diego, 1987.
11. G. Hinton and T.J. Sejnowski, "Optimal Perceptual Inference", in *Proceedings of the IEEE Conference on Computer Vision*, 1983, p. 458.
12. P. Peretto, "Collective Properties of Neural Networks:

- A Statistical Physics Approach", *Biological Cybernetics* 50, pp. 51-62, 1984.
13. K. Venugopal, A. Pandya and R. Sudhakar, "ALOPEX Algorithms for Adaptive Control of Dynamical Systems", in *Proceedings on Neural Networks*, Vol. II, 1992, p. 875.
 14. K. Narendra and K. Parthasarathy, "Identification and Control of Dynamical Systems Using Neural Networks", *IEEE Transactions on Neural Networks*, Vol. 1, No. 1, pp. 4-27, March, 1990.
 15. A. Karakasoglu, S.I. Sudharsanan and M.K. Sundareshan, "Identification and Decentralized Adaptive Control of Robotic Manipulators Using Dynamical Neural Networks", *Proceedings of the 1991 Joint International Conference on Neural Networks (IJCNN-91)*, Seattle, Washington, July, 1991.
 16. R. Williams and D. Zipser, "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks", *Neural Computation* 1, pp. 270-280, 1989.
 17. B. Pearlmutter, "Learning State Space Trajectories in Recurrent Neural Networks", *Neural Computation* 1 pp. 263-269, 1989.
 18. N. Toomarian and J. Barhen, "Learning a Trajectory Using Adjoint Functions and Teacher Forcing", *Neural Networks*, Vol. 5, 1992, pp. 473-484.
 19. E. Oja, "A Simplified Neuron Model As a Principal Component Analyzer", *Journal of Mathematical Biology* 15, pp. 267-273, 1982.
 20. T. Sanger, "Optimal Unsupervised Learning in a Single-Layer Linear Feed-Forward Network", *Neural Networks* 2, pp. 459-473, 1989.
 21. R. Linsker, "From Basic Network Principles to Neural Architecture", *Proceedings of the National Academy of Sciences USA* 83, pp. 7508-7512, 1986.
 22. K. Narendra and M. Thathachar, *Learning Automata, an Introduction*, Addison Wesley, 1989.
 23. M. Tsetlin, "On the Behavior of Finite Automata in Random Media", *Automation and Remote Control*, Vol. 22, pp. 1210-1219, 1962. (Originally in *Avtomatika i Telemekhanika*, Vol. 22, 1961)

24. K. Narendra, E.A. Wright and L.G. Mason, "Application of Learning Automata to Teletraffic Routing and Management", *IEEE Transactions on Systems, Man and Cybernetics*, SMC-7, pp. 785-792, 1977.
25. K. Narendra and P. Mars, "The Use of Learning Automata in Telephone Traffic Routing - A Methodology", *Automatica*, Vol. 19, pp. 495-502, 1983.
26. M. Meybodi. and S. Lakshmivarahan, "A Learning Approach to Priority Assignment in Two Class M/M/1 Queueing System with Unknown Parameters", in *Proceedings of the Third Yale Workshop on Applications of Adaptive Systems Theory*, pp. 1-20, 1983.
27. P. Srikanthakumar, "A Simple Learning Scheme for Priority Assignment at a Single-Server Queue", *IEEE Transactions on Systems, Man and Cybernetics*, SMC-16, pp. 751-754, 1986.
28. A. Hashim, S. Amir and P. Mars, "Application of Learning Automata to Data Compression", *Adaptive Learning and Systems*, New York: Plenum Press, 1986, pp. 229-234.
29. J. Hertz, A. Krogh and R. Palmer, *Introduction to the Theory of Neural Computation*, Redwood City, Calif: Addison-Wesley, 1991.
30. T. Khanna, *Foundations of Neural Networks*, Reading, Mass: Addison-Wesley, 1991.
31. R. Bartle and D. Sherbet, *Introduction to Real Analysis*, New York: John Wiley and Sons, 1992.
32. N. Baba, *New Topics in Learning Automata Theory and Applications*, Vol. 71, Berlin: Springer-Verlag, 1984.
33. Y. Ichikawa and T. Sawa, "Neural Network Application for Direct Feedback Controllers", *IEEE Transactions on Neural Networks*, Vol3, No. 2, pp. 224-232, March, 1992.
34. F. Cellier, *Continuous System Modeling*, New York: Springer-Verlag, 1991.
35. C. Gerald and P.O. Wheatley, *Applied Numerical Analysis*, Reading, Mass.: Addison-Wesley, 1989.
36. B. Widrow and F.W. Smith, "Pattern Recognizing Control Systems", presented at *1963 Computer and Information Symposium Proceedings*, Washington, D.C., 1964.

37. B. Widrow, "The Original Adaptive Neural Net Broom-Balancer", in *International Symposium on Circuits and Systems*, 1987, p.351.
38. A. Barto, R. Sutton and C. Anderson, "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems", *IEEE Transactions on Systems, Man and Cybernetics*, Vol. SMC-13, pp. 834-846, September, 1983.
39. D. Michie and R. Chambers, "BOXES: An Experiment in Adaptive Control", *Machine Intelligence*, Edinburgh: Oliver and Boyd, 1968, pp. 137-152.
40. C. Anderson, "Strategy Learning with Multilayer Connectionist Representations", GTE Laboratories, Waltham, Mass., Report TR87-509.3, June, 1987.
41. C. Anderson, "Learning to Control an Inverted Pendulum Using Neural Networks", *IEEE Control Systems Magazine*, pp. 31-37, April, 1989.
42. D. Psaltis, A. Sideris and A. Yamamura, "A Multilayered Neural Network Controller", *IEEE Control Systems Magazine*, Vol. 8, pp.17-21, 1988.
43. M. Kawato, Y. Uno, M. Isobe and R. Suzuki, "Hierarchical Neural Network Model for Voluntary Movement with Application to Robotics", *IEEE Control Systems Magazine*, Vol. 8, pp. 8-15, 1988.
44. W. Miller, R. Hewes, F. Glanz and L. Kraft, "Real-time Control of an Industrial Manipulator Using a Neural Network, based Learning Controller", *IEEE Journal of Robotics Research*, Vol 6.2, pp. 84-98, 1987.
45. I. Muhsin, M. Sundareshan, S. Sudharsanan and A. Karakasoglu, "Adaptive Excitation and Governor Control of Synchronous Generators Using Multilayer Recurrent Neural Networks", *Proceedings of the 31st IEEE Conference on Decision and Control*, Tucson, Arizona, December, 1992, pp. 589-591.
46. K. Narendra and K. Parthasarathy, "Identification and Control of Dynamical Systems Using Neural Networks", *IEEE Transactions on Neural Networks*, Vol. 1, No. 1, pp. 4-27, March, 1990.
47. M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, Cambridge, Mass: M.I.T. Press, 1969.

48. B. Widrow and M. Hoff, "Adaptive Switching Circuits", in *IRE WESCON Convention Record*, 1960.
49. C. von der Malsburg, "Self-Organization of Orientation Sensitive Cells in the Striate Cortex", *Kybernetik* 14, pp. 85-100, 1973.
50. K. Fukushima, "Cognitron: A Self-Organizing Multilayered Neural Network", *Biological Cybernetics* 20, pp. 121-136, 1975.
51. S. Grossberg, "Adaptive Pattern Classification and Universal Recoding", *Biological Cybernetics* 23, pp. 121-134, 1976.
52. T. Kohonen, "Self-Organized Formation of Topologically Correct Feature Maps", *Biological Cybernetics* 43, pp. 59-69, 1982.
53. M. Cohen and S. Grossberg, "Absolute Stability of Global Pattern Stability and Parallel Memory Storage by Competitive Neural Networks", *IEEE Transactions on Systems, Man and Cybernetics* 13, pp. 815-826, 1983.
54. J. Hopfield, "Neurons With Graded Responses Have Collective Computational Properties Like Those of Two-State Neurons", *Proceedings of the National Academy of Sciences, USA* 81, pp. 3088-3092, 1984.
55. A. Karaksoglu, S. Sudharsanan and M. Sundareshan, "Neural Network-Based Identification and Control of Nonlinear Systems: A Novel Dynamical Network Architecture and Training Policy", *Proceedings of the 30th IEEE Conference on Decision and Control*, Brighton, England, December, 1991, pp. 180-182.
56. S. Sudharsanan, I. Muhsin and M. Sundareshan, "Self-Tuning Adaptive Control of Multi-Input Multi-Output Nonlinear Systems Using Multilayer Recurrent Neural Networks With Application to Synchronous Power Generators", *Proceedings of 1993 IEEE International Conference on Neural Networks*, San Francisco, California, March, 1993, pp. 1307-1311.