Master's Thesis

New Neural Network Design for Approximate Dynamic Programming and Optimal Multiuser Detection

by Xiaozhong Pang
Advisor: John S. Baras

CSHCN M.S. 98-1
(ISR M.S. 98-1)

ABSTRACT

Title of Thesis: NEW NEURAL NETWORK DESIGN FOR APPROXIMATE DYNAMIC PROGRAMMING AND OPTIMAL MULTIUSER DETECTION

Degree candidate: Xiaozhong Pang

Degree and year: Master of Science, 1997

Thesis directed by: Professor John S. Baras
Department of Electrical Engineering and
Institute for Systems Research

In this thesis we demonstrate that a new neural network design can be used to solve a class of difficult function approximation problems which are crucial to the field of approximate dynamic programming(ADP). Although conventional neural networks have been proven to approximate smooth functions very well, the use of ADP for problems of intelligent control or planning requires the approximation of functions which are not so smooth. As an example, this thesis studies the problem of approximating the J function of dynamic programming applied to the task of navigating mazes, in general, without the need to learn each individual maze. Conventional neural networks, like multi-layer perceptrons(MLPs), cannot learn this task. But a new type of neural networks, simultaneous recurrent networks(SRNs), can accomplish the required learning as demonstrated by

successful initial tests. In this thesis we investigate also the ability of recurrent neural networks to approximate MLPs and vice versa. Moreover, we present a comparison between using SRNs and MLPs to implement the optimal CDMA multiuser detector (OMD). This example is intended to demonstrate that SRNs can provide fast suboptimal solutions to hard combinatorial optimization problems, and achieve better bit-error-rate (BER) performance than MLPs.

NEW NEURAL NETWORK DESIGN FOR APPROXIMATE DYNAMIC PROGRAMMING AND OPTIMAL MULTIUSER DETECTION

by

Xiaozhong Pang

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland at College Park in partial fulfillment
of the requirements for the degree of
Master of Science
1997

Advisory Committee:

Professor John S. Baras, Chairman/Advisor
Professor Carlos A. Berenstein
Professor Leandros Tassiulas

ACKNOWLEDGMENTS

I would like to express my deep gratitude to Professor John S. Baras, my advisor, for his support to my graduate study and research at the Institute for Systems Research in the University of Maryland at College Park. The work for this thesis could not have been finished without Professor Baras' supervision.

I am very grateful to Dr. Paul J. Werbos for his help, suggestions and encouragement while I was doing the research on this thesis.

I would equally thank Professor Carlos A. Berenstein and Professor Leandros Tassiulas for their support and valuable comments that led to generating the final version of this thesis.

My thanks also go to Bernard A. Frankpitt and Wei Luo, with whom I discussed some problems related to this thesis, and all the people who work in the Systems Engineering and Integration Laboratory, where I finished most of my research work.

The financial support provided by the National Science Foundation Engineering Research Centers Program Grant NSFD CDR 8803012 through the Institute for Systems Research (ISR) and by NASA under contract NCC3-528 through the Center for Satellite and Hybrid Communication Networks (CSHCN) are most greatly appreciated.

Finally, and most importantly, I wish to thank Chengyin Ye, for his love, help and understanding.

TABLE OF CONTENTS

LIST OF FIGURES **vii**

1 Introduction **1**

1.1 Goals/Objectives 1

1.2 Background 2

1.3 Summary and Organization of the Thesis 3

2 Motivation **8**

2.1 What is ADP and J Function? 10

2.2 Intelligent Control and Robust Control 12

2.3 Importance of the SRN to ADP 13

3 Alternative Forms of Recurrent Networks **15**

3.1 Purpose 15

3.2 Structure of Discrete-Time Recurrent Networks 17

3.3 Training of SRNs and TLRNs 21

3.3.1 Backpropagation through time(BTT) 23

3.3.2 Truncation for SRN 27

3.3.3 Simultaneous Backpropagation 30

3.3.4 Error Critic 31

3.3.5 Forward Propagation 32

4 Three Test Problems and Details on Architecture and Learning Procedures 33

4.1 Net A/Net B 33

4.2 The Maze Problem 35

4.3 Details for the Net A/Net B Problem 38

4.4 Weight-sharing and Cellular Architecture 38

4.4.1 What is Weight-sharing? 38

4.4.2 Lie Group Symmetry and Weight-sharing 39

4.4.3 How We implemented Weight-sharing 40

4.5 Adaptive Learning Rate 43

4.6 An SRN receiver for OMD 45

4.6.1 Multiuser Detection Schemes 46

4.6.2 Neural net receiver 49

4.6.3 An SRN receiver 52

5 Simulation Results and Conclusions 53

5.1 Results for the Net A/Net B Problem 53

5.2 Results for the Maze Problem 56

5.3 Comparison of the performance of MUD using OMD, SRN and MLP 57

5.4 Conclusions 61

A Appendix: The program of the maze problem using SRN trained by BTT **64**

LIST OF FIGURES

1.1 What is supervised learning? 1

3.1 Recurrent networks 16

3.2 Time lagged recurrent network (TLRN) 18

3.3 Simultaneous recurrent network (SRN) 18

3.4 Time delayed neural network (TDNN) 21

3.5 Types of SRN Training 22

3.6 Error Critics 23

3.7 Backpropagation through time(BTT) 24

3.8 BTT for TLRN 26

3.9 BTT for SRN 26

3.10 Truncation 29

4.1 Net A/Net B 34

4.2 Desired J function of a maze 37

4.3 General Idea of the Cellular Network 41

4.4 Inputs, Outputs and Memory of Each Cell 42

4.5 The general structure of multiuser detection(MUD) 47

4.6 A neural network receiver for multiuser demodulation 49

4.7 Hopfield neural network 50

5.1 The MLP learned the SRN 54

5.2 The SRN learned the MLP 54

5.3 The last 1000 trials of Figure 5.1 55

5.4 The last 1000 trials of Figure 5.2 55

5.5 J function as predicted by SRN-BTT(I) 56

5.6 J function as predicted by SRN-Truncation(I) 57

5.7 Error curve of itJ function as predicted by SRN-BTT(II) 58

5.8 Error curve of itJ function as predicted by SRN-BTT(III) 58

5.9 Error curve of J function as predicted by SRN-Truncation(II) . . 59

5.10 Error curve of J function as predicted by SRN-Truncation(III) . . 59

5.11 Error curve of J function as predicted by MLP(I) 60

5.12 Error curve of J function as predicted by MLP(II) 60

5.13 Bit error rate versus E_1/N for 5-user channel with the SNR of the other users fixed at 8dB, where solid line is the OMD, the dash line is the SRN and the dash-dot line is the MLP. 61

5.14 Bit error rate versus E_i/E_1 for 5-user channel with the SNR of user 1 fixed at 8dB, where the solid line is OMD, the dash-dot line is the SRN and the dotted line is the MLP. 62

CHAPTER 1

Introduction

1.1 Goals/Objectives

This thesis has three goals:

First, to demonstrate the value of a new class of neural networks which provide a crucial component needed for brain-like intelligent control systems for the future.

Second, to demonstrate that this new kind of neural networks provide better function approximate ability for use in more ordinary kinds of neural network applications for supervised learning.

Third, to demonstrate some practical implementation techniques necessary to make this kind of networks actually work in practice.

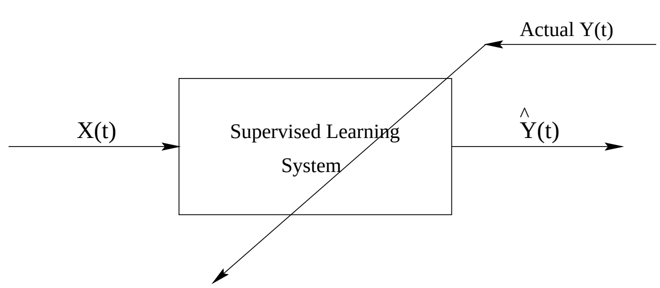

Figure 1.1: What is supervised learning?

1.2 Background

At present, in the neural network field, perhaps 90% of neural network applications involve the use of neural networks designed to perform a task called supervised learning (Figure 1.1). Supervised learning is the task of learning a nonlinear function which may have several inputs and several outputs based on some examples of the function. For example, in character recognition, the inputs may be an array of pixels seen from a camera. The desired outputs of the network may be a classification of characters being seen. Another example would be intelligent sensing in the chemical industry where the inputs might be spectral data from observing a batch of chemicals, and the desired outputs would be the concentrations of the different chemicals in the batch. The purpose of this application is to predict or estimate what is in the batch without the need for expensive analytical tests.

The work in this thesis will focus totally on certain tasks in supervised learning. Even though existing neural networks can be used in supervised learning, there can be performance problems depending on what kind of function the network is trying to learn. Many people have proven many theorems which show that neural networks, fuzzy logic, Taylor theories and other function approximations have a universal ability to approximate functions on the condition that the functions have certain properties and that there is no limit on the complexity of the approximation. In practice, many approximation schemes become useless when there are many input variables because the required complexity grows at an exponential rate.

For example, one way to approximate a function would be to construct a table of the values of the function at certain points in the space of possible inputs.

Suppose there are 30 input variables and we consider 10 possible values of each input. In that case, the table must have 10^{30} numbers in it. This is not useful in practice for many reasons. Actually, however, many popular approximation methods like radial basis functions (RBF) neural networks are similar in spirit to a table of values.

In the field of supervised learning, Andrew Barron [30] proved several function approximation theorems which are much more useful in practice. He has proven that the most popular form of neural networks, the multi-layer perceptron(MLP), can approximate any smooth function. Unlike the case with the linear basis functions (like RBF and Taylor series), the complexity of the network does not grow as rapidly as the number of input variables grows.

Unfortunately there are many practical applications where the functions to be approximated are not smooth. In some cases, it is good enough just to add extra layers to an MLP [1] or to use a generalized MLP [2]. However, there are some difficult problems which arise in fields like intelligent control or image processing or even stochastic search, where feed-forward networks do not appear powerful enough.

1.3 Summary and Organization of the Thesis

The main goal of this thesis is to demonstrate the capability of a different kind of supervised learning system based on a kind of recurrent networks called simultaneous recurrent networks (SRNs). In the next chapter we explain why this kind of improved supervised learning system will be very important to intelligent control and to approximate dynamic programming. In effect, this work on supervised learning, is the first step in a multi-step effort to build more brain-like intelligent systems. The next step would be to apply the SRN to static optimization problems, and then to integrate the SRNs into large systems for ADP.

Even though intelligent control is the main motivation for this work, the work may be useful for other areas as well. For example, in zip code recognition, $AT\&T$ [3] has demonstrated that feed-forward networks can achieve a high level of accuracy in classifying individual digits. However, $AT\&T$ and others still have difficulty in segmenting the total zip codes into individual digits. Research on human vision by von der Malsburg [4] and others has suggested that some kinds of recurrency in neural networks are crucial to their abilities in image segmentation and binocular vision. Furthermore, researchers in image processing like Laveen Kanal [41] have showed that iterative relaxation algorithms are necessary to achieve even moderate success in such image processing tasks. Conceptually the SRN can learn an optimal iterative algorithm, but the MLP cannot represent any iterative algorithms. In summary, though we are most interested in brain-like intelligent control, the development of SRNs could lead to very important applications in areas such as image processing in the future.

The network described in this thesis is unique in several respects. However, it is certainly not the first serious use of a recurrent neural network. In Chapter 3 of this thesis we provide a review of the existing literature on recurrent networks. We also describe the relationship between this new design and other designs in the literature. Roughly speaking, the vast bulk of research in recurrent networks has been academic research using designs based on ordinary differential equations (ODE) to perform some tasks very different from supervised learning — tasks like clustering, associative memory and feature extraction. The simple Hebbian learning methods[13] used for those tasks do not lead to the best performance in supervised learning. Many engineers have used another type of recurrent network , the time lagged recurrent network (TLRN), where the recurrence is used to provide memory of past time periods for use in forecasting the future. However, this type of recurrence cannot provide the iterative analysis capability mentioned above. There are very few research reports about SRNs, a type of recurrent network designed to minimize error and learn an optimal iterative approximation to a function. This is certainly the first use of SRNs to learn a J function from dynamic programming which will be explained in more detail in Chapter 2. This may also be the first empirical demonstration of the need for advanced training methods to permit SRNs to learn difficult functions.

In Chapter 4 we explain in more detail the three test problems we have used for the SRN and the MLP, as well as the details of the architecture and of the learning procedure.

The first test problem was used mainly as an initial test of a simple form of SRNs. In this problem, we tried to test the hypothesis that an SRN can always learn to approximate a randomly chosen MLP, but not vice versa. Although our results are consistent with that hypothesis, there is room for more extensive work in the future, such as experiments with different sizes of neural networks and more complex statistical analysis.

The main test problem in this work was the problem of learning the J function value searching of dynamic programming. For a maze navigation problem, many neural network researchers have written about neural networks which learn an optimal policy of action for one particular maze [5]. This thesis addresses the more difficult problem of training a neural network to input a picture of a maze and output the J function for this maze. When the J function is known, it is a trivial local calculation to find the best direction of movement. This kind of neural network should not require retraining whenever a new maze is encountered. Instead it should be able to look at the maze and immediately "see" the optimal strategy. Training such a network is a very difficult problem which has never been solved in the past with any kind of neural network. Also it is typical of the challenges one encounters in true intelligent control and planning. This thesis identifies a working solution to this problem for the first time. Now that a system is working on a very simple form for this problem, it would be possible in the future to perform many tests of the ability of this system to generalize its success to many mazes.

In order to solve the maze problem, it was not sufficient to use an SRN. There are many choices to make when implementing the general idea of SRNs or MLPs. In Chapter 4 we describe also in detail how these choices were made in this work. The most important choices were:

1. Both for the MLP and for the feed-forward core of the SRN we used the generalized MLP design [2], which eliminates the need to decide on the number of layers.

2. For the maze problem, we used a cellular or weight-sharing architecture which exploits the spatial symmetry of the problem and reduces dramatically the number of weights. In effect we solved the maze problem using only five distinct neurons. There are interesting parallels between this network and the hippocampus of the human brain.

3. For the maze problem, an adaptive learning rate (ALR) procedure was

used to prevent oscillation and ensure convergence.

4. Initial values for the weights and the initial input vector for the SRN were chosen essentially at random, by hand. In the future, more systematic methods are available. But this method was sufficient for success in the present case.

The third test problem is to use SRNs for implementing the optimum multiuser detector (OMD) in code division multiple access (CDMA). This is an example to use SRNs for static optimization, and show that SRNs have better functional approximate abilities than MLPs, as we have shown in the NetA/NetB and maze problem. In this problem, we tried to use SRNs to approach the optimal bit-error-rate (BER) performance while avoiding the NP-completeness that characterizes the OMD computations, i.e. to reduce the computational complexity.

Finally in Chapter 5 we discuss the simulation results in more detail, give the conclusions of this thesis and mention some possibilities for future work.

CHAPTER 2

Motivation

In this chapter we will explain the significance of this work. As discussed above, the thesis shows how to use a new type of neural network in order to achieve better function approximation than what is available from the types of neural networks which are popular today. In this chapter we explain why better function approximation is important to approximate dynamic programming (ADP), intelligent control and understanding of the brain. Image processing and other applications have already been discussed in the Introduction. These three topics — ADP, intelligent control and understanding of the brain — are all closely related to each other and provide the original motivation for the work of this thesis.

The purpose of this thesis is to make a core contribution to developing the most powerful possible system for intelligent control.

In order to build the best intelligent control systems, we need to combine the most suitable mathematics together with some understanding of natural intelligence in the brain. There is currently strong research interest in intelligent control world-wide. Some control systems which are called intelligent are actually very quick and simple designs. There are many researchers who attempt to move step by step to add intelligence into control, but a step-by-step approach may not be enough by itself.

Sometimes, to achieve a complex and difficult goal, it is necessary to have a plan; thus some parts of the intelligent control community have developed a more systematic vision or plan about how it could be possible to achieve real intelligent control. First, one must think about the question of what is intelligent control. Then, instead of trying to answer this question in one step, we try to develop a plan to reach the design. Actually there are two questions:

1. How could we build an artificial system which replicates the main capabilities of brain-like intelligence, somehow unified together as they are unified together in the brain?

2. How can we understand what are the capabilities in the brain and how they are organized in a functional engineering view? i.e. how are those circuits in the human brain arranged to learn how to perform different tasks?

It would be best to understand how the human brain works before building an artificial system. However, at the present time, our understanding of the brain is limited. But at least we know that local recurrence plays a critical role in the higher parts of the human brain [6][7][8][4].

Another reason for using SRNs is that SRNs can be very useful in ADP mathematically. Now we will discuss what ADP can accomplish for intelligent control and understanding of the brain.

The remainder of this chapter will address three questions in order:

1. What is ADP?

2. What is the importance of ADP to intelligent control and understanding of

the brain?

3. What is the importance of SRNs to ADP?

2.1 What is ADP and J Function?

To explain what is ADP, let us consider the original Bellman equation[9]:

$$J(R(t)) = \max_{u(t)}(U(R(t), u(t)) + < J(R(t+1)) >)/(1+r) - U_0 \tag{2.1}$$

where r and U_0 are constants that are used only in infinite-time-horizon problems, and where the angle brackets refer to expectation value. In this thesis we actually use:

$$J(R(t)) = \max_{u(t)}(U(R(t), u(t)) + < J(R(t+1)) >) \tag{2.2}$$

since the maze problem does not involve an infinite time-horizon.

Instead of solving for the value of J in every possible state, $R(t)$, we can use a function approximation method like neural networks to approximate the J function. This is called approximate dynamic programming(ADP). This thesis is not employing "true ADP" because in "true ADP" we do not know what the J function is and must therefore use indirect methods to approximate it. However, before we try to use SRNs as a component of an ADP system, it makes sense to first test the ability of an SRN to approximate a J function, in principle.

Now we will try to explain what is the intuitive meaning of the Bellman equation (Equation(2.1)) and the J function according to the treatment of dynamic programming given in [2].

To understand ADP, one must first review the basics of classical dynamic programming, especially the versions developed by Howard [28] and Bertsekas [41]. Classical dynamic programming is the only exact and efficient method to compute the optimal control policy over time, in a general nonlinear stochastic environment. The only reason to approximate it is to reduce computational cost, so as to make the method affordable (feasible) across a wide range of applications. In dynamic programming, the user supplies a utility function which may take the form $U(R(t), u(t))$ — where the vector R is a representation or estimate of the state of the environment (i.e. the state vector) — and a stochastic model of the plant or environment. Then "dynamic programming" (i.e. solution of the Bellman equation) gives us back a secondary or strategic utility function $J(R)$. The basic theorem is that maximizing $U(R(t), u(t)) + < J(R(t+1)) >$ yields the optimal strategy, the policy which will maximize the expected value of U added up over all future time. Thus dynamic programming converts a difficult problem in optimizing over many time intervals into a straightforward problem in short-term maximization. In classical dynamic programming, we find the exact function J which exactly solves the Bellman equation. In ADP, we learn a kind of "model" of the function J; this "model" is called a "Critic" [9]. Alternatively, some methods learn a model of the derivatives of J with respect to the variables R_i ; these correspond to Lagrange multipliers, λ_i , and to the "price variables" of microeconomic theory. Some methods learn a function related to J, as in the Action-Dependent Adaptive Critic (ADAC) [29].

2.2 Intelligent Control and Robust Control

To understand the human brain scientifically, we must have some suitable mathematical concepts. Since the human brain makes decisions like a control system, it is an example of an intelligent control system. Neuroscientists do not yet understand the general ability of the human brain to learn to perform new tasks and solve new problems even though they have studied the brain for decades. Some people compare the past research in this field to what would happen if we spent years to study radios without knowing the mathematics of signal processing.

We first need some mathematical ideas of how it is possible for a computing system to have this kind of capability based on distributed parallel computation. Then we must ask what are the most important abilities of the human brain which unify all of its more specific abilities in specific tasks. It would be seen that the most important ability of the brain is the ability to learn over time how to make better decisions in order to better maximize the goals of the organism. The natural way to imitate this capability in engineering systems is to build systems which learn over time how to make decisions which maximize some measure of success or utility over future time. In this context, dynamic programming is important because it is the only exact and efficient method for maximizing utility over future time. In the general situation, where random disturbances and nonlinearity are expected, ADP is important because it provides both the learning capability and the possibility of reducing computational cost to an affordable level. For this reason, ADP is the only approach we have available to imitate the learning ability of the brain.

The similarity between some ADP designs and the circuitry of the brain has been discussed at length in [10] and [11]. For example, there is an important structure in the brain called the limbic system which performs some kind of evaluations or reinforcement functions, very similar to the functions of the neural networks that must approximate the J function of dynamic programming. The largest part of the limbic system, called the hippocampus, is known to possess a higher degree of local recurrence [8].

In general, there are two ways to make classical controllers stable despite great uncertainty about parameters of the plant to be controlled. For example, in controlling a high speed aircraft, the location of the center of the gravity is not known. The center of gravity is not known exactly because it depends on the cargo of the air plane and the location of the passengers. One way to account for such uncertainties is to use adaptive control methods. We can get similar results, but more assurance of stability in most cases [16] by using related neural network methods, such as adaptive critics with recurrent networks. It is like adaptive control but more general. There is another approach called robust control or H_∞ control, which tries to design a fixed controller which remains stable over a large range in parameter space. Baras and Patel [31] have solved the general problem of H_∞ control for general partially observed nonlinear plants with set valued dynamics and disturbances. They have shown that this problem reduces to a problem in nonlinear, stochastic optimization. Adaptive dynamic programming makes it possible to solve large scale problems of this type.

2.3 Importance of the SRN to ADP

ADP systems already exist which perform relatively simple control tasks like stabilizing an aircraft as it lands under windy conditions [12]. However, such tasks do not really represent the highest level of intelligence or planning. True intelligent control requires the ability to make decisions when future time periods will follow a complicated, unknown path starting from the initial state. One example of a challenge for intelligent control is the problem of navigating a maze which we will discuss in Chapter 4. A true intelligent control system should be able to learn this kind of task. However, the ADP systems in use today could never learn this kind of task. They use conventional neural networks to approximate the J function. Because the conventional MLP cannot approximate such a J function, we may deduce that ADP systems constructed only from MLPs will never be able to display this kind of intelligent control. Therefore, it is essential that we can find a kind of neural network which can perform this kind of task. As we will show, the SRN can fill this crucial gap. There are additional reasons for believing that the SRN may be crucial to intelligent control as discussed in chapter 13 of [9].

CHAPTER 3

Alternative Forms of Recurrent Networks

3.1 Purpose

There is a huge literature on recurrent networks. Biologists have used many recurrent models because the existence of recurrence in the brain is obvious. However, most of the recurrent networks implemented so far have been classic style recurrent networks, as shown on the left hand of Figure 3.1. Most of these networks are formulated from ordinary differential equation (ODE) systems. Usually their learning is based on a restricted concept of Hebbian learning. Originally in the neural network field, the most popular neural networks were recurrent networks like those which Hopfield [14] and Grossberg [15] used to provide associative memory.

Associative memory networks can actually be applied to supervised learning. But in actuality their capabilities are very similar to those of look-up tables and radial basis functions. They make predictions based on similarity to previous examples or prototypes. They do not really try to estimate general functional relationships. As a result these methods have become unpopular in practical

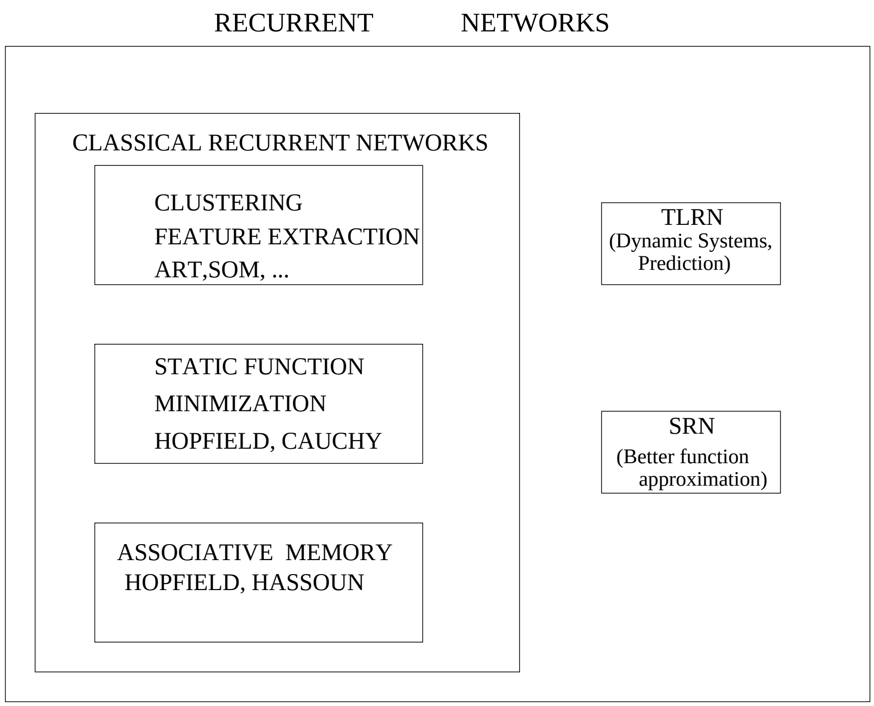

Figure 3.1: Recurrent networks

applications of supervised learning. The theorems of Barron [30] discussed in the Introduction show that MLPs do provide better function approximation than do simple methods based on similarity.

There has been substantial progress in the past few years in developing new associative memory designs. Nevertheless, the MLP is still better for the specific task of function approximation which is the focus of this thesis.

Actually the problem of static optimization will be considered more in future stages of this research. When people use the classic Hopfield networks for static optimization, they specify all the weights and connections in advance [14]. This has limited the success of this kind of networks for large scale problems where it is difficult to guess the weights. With the SRN we have methods to train the weights in that kind of structure. Thus the guessing is no longer needed.

There have also been researchers using ODE neural networks who have tried to use training schemes based on a minimization of error instead of Hebbian approaches. However, in practical applications of such networks, it is important to consider the clock rates of computation and data sampling. For that reason, it is both easier and better to use error minimizing designs based on discrete time rather than ODEs.

In a similar way, classic recurrent networks have been used for tasks like clustering, feature extraction and static function optimization. But these are different problems from the ones we are trying to solve here.

3.2 Structure of Discrete-Time Recurrent Networks

If the importance of neural networks is measured by the number of words published, then the classic networks dominate the field of recurrent networks. However, if the value is measured based on the economic value of practical applications, then the field is dominated by time-lagged recurrent networks (TLRNs). The purpose of the TLRNs is to predict or classify time-varying systems using recurrence as a way to provide memory of the past. The SRNs have some relation with the TLRNs but it is designed to perform a fundamentally different task. The SRNs use recurrence to represent more complex relationships between one input vector $X(t)$ and one output $Y(t)$ without consideration of the other times t. Figure 3.2 and Figure 3.3 show us more details about the TLRNs and the SRNs.

In control applications, $u(t)$ represents the control variables which we use to control the plant. For example, if we design a controller for a car engine, the $X(t)$ variables are the data we get from our sensors. The $u(t)$ variables would include

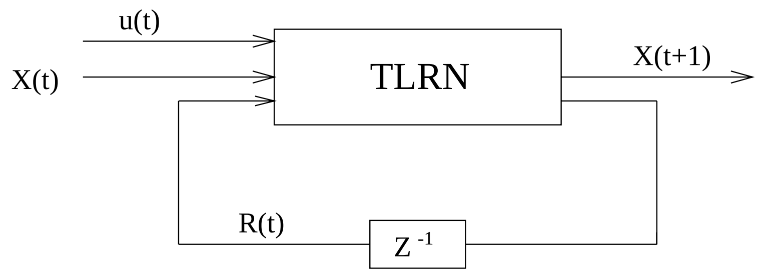

Figure 3.2: Time lagged recurrent network (TLRN)

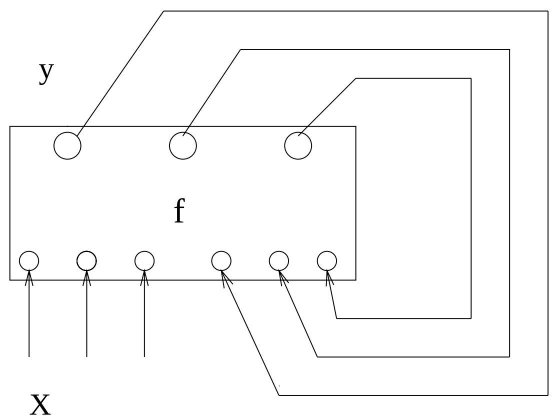

Figure 3.3: Simultaneous recurrent network (SRN)

the valve settings which we use to try to control the process of combustion. The $R(t)$ variables provide a way for the neural networks to remember past time cycles, and to implicitly estimate important variables which cannot be observed directly. In fact, the application of TLRNs to automobile control is the most valuable application of recurrent networks ever developed so far.

A simultaneous recurrent network (Figure 3.3) is defined as a mapping:

$$\hat{Y}(t) = F(X(t), W) \tag{3.1}$$

which is computed by iterating over the following equation:

$$y^{(n+1)}(t) = f(y^{(n)}(t), X(t), W) \tag{3.2}$$

where f is some sort of feed-forward network or system, and $\hat{Y}$ is defined as:

$$\hat{Y}(t) = \lim_{n \to \infty} y^{(n)}(t) \tag{3.3}$$

When we use $\hat{Y}$ in this thesis, we use $n = 20$ instead of ∞ here.

In Figure 3.3, the outputs of the neural network come back again as inputs to the same network. However, in concept there is no time delay. The inputs and outputs should be simultaneous. That is why it is called a simultaneous recurrent network (SRN). In practice, of course, there will always be some physical time delay between the outputs and the inputs. However if the SRN is implemented in fast computers, this time delay may be very small compared to the delay between different frames of input data.

In Figure 3.3, X refers to the input data at the current time frame t. The vector y represents the temporary output of the network, which is then recycled as an additional set of inputs to the network. At the center of the SRN is actually the feed-forward network which implements the function f. (In designing an SRN, you can choose any feed-forward network or system as you like. The function f simply describes which network you use). The output of the SRN at any time t is simply the limit of the temporary output y.

In Equations (3.1) and (3.2), notice that there are two integers — n and t — which could both represent some kind of time. The integer t represents a ***slower*** kind of time cycle, like the delay between frames of incoming data. The integer n represents a ***faster*** kind of time, like the computing cycle of a fast electronic chip. For example, if we build a computer to analyze images coming from a movie camera, "t" and "$t+1$" represent two successive incoming pictures with a movie camera. There are usually only 32 frames per second. (In the human brain, it seems that there are only about 10 frames per second coming into the neocortex.) But if we use a fast neural network chip, the computational cycle — the time between "n" and "$n+1$" — could be as small as a microsecond.

In actuality, it is not necessary to choose between time-lagged recurrency (from t to $t+1$) and simultaneous recurrency (from n to $n+1$). It is possible to build a hybrid system which contains both types of recurrency. This could be very useful in analyzing data like movie pictures, where we need both memory and some ability to segment the images. [9] discusses how to build such a hybrid system. However, before building such a hybrid system, we must first learn to make SRNs work by themselves.

Finally, we note that TLRNs are not the only kind of neural networks used in predicting dynamical systems. Even more popular are the time delayed neural networks (TDNNs), shown in Figure 3.4. The TDNNs are popular because they are easy to use. However, they are less capable, in principle, because they have no ability to estimate unknown variables. They are especially weak when some of these variables change slowly over time and require memory which persists over long time periods. In addition, the TLRNs fit the requirements of ADP directly, while the TDNNs do not [9][16].

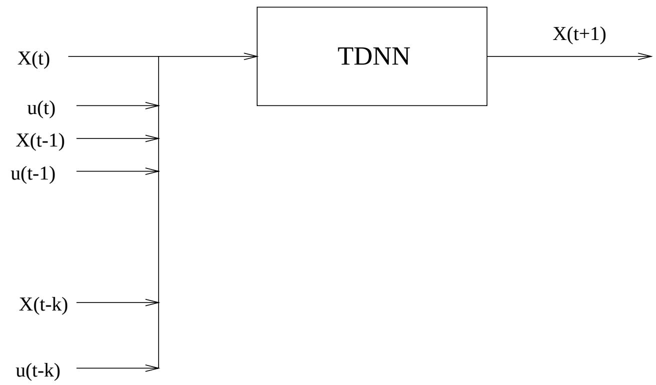

Figure 3.4: Time delayed neural network (TDNN)

3.3 Training of SRNs and TLRNs

There are many types of training that have been used for recurrent networks. Different types of training give rise to different kinds of capabilities for different tasks. For the tasks which we have described for the SRNs and the TLRNs, all proper forms of training involve some calculation of the derivatives of error with respect to the weights. Usually after these derivatives are known, the weights are adapted according to a simple formula as follows:

$$newW_{i,j} = oldW_{i,j} - LR * \frac{\partial Error}{\partial W_{i,j}} \tag{3.4}$$

where LR is called the *learning rate*.

There are five main ways to train SRNs, all based on different methods for calculating or approximating the derivatives. Four of these methods can also be used with TLRNs. Some can be used for control applications. But the details of those applications are beyond the scope of this thesis. These five types of training are listed in Figure 3.5. For this thesis, we have used two of these methods: Backpropagation through time (BTT) and Truncation.

The five methods are:

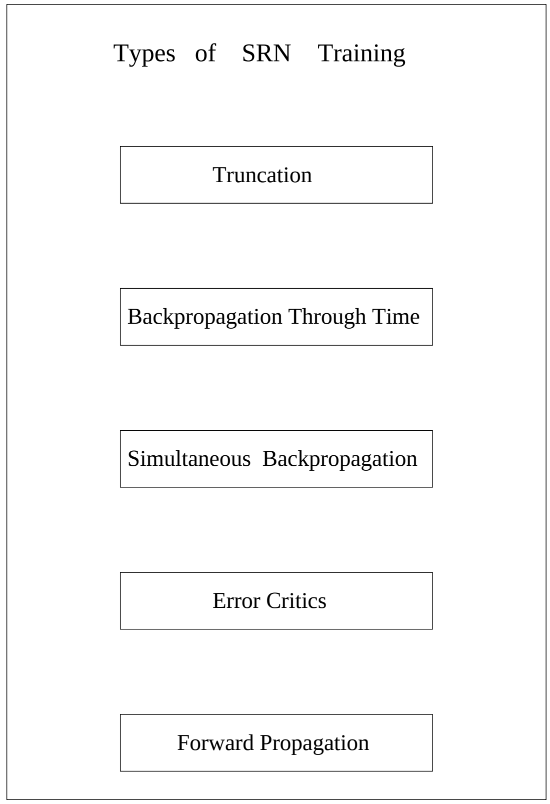

Figure 3.5: Types of SRN Training

1. ***Backpropagation through time (BTT)***. This method and forward propagation are the two methods which calculate the derivatives exactly. BTT is also less expensive than forward propagation.

2. ***Truncation***. This is the simplest and least expensive method. It uses only one simple pass of backpropagation through the last iteration of the model. Truncation is probably the most popular method used to adapt SRNs even though the people who use it the most, call it just ordinary backpropagation.

3. ***Simultaneous backpropagation***. This is more complex than truncation, but it still can be used in real time learning. It calculates derivatives which are exact in the neighborhood of equilibrium but it does not account for the details of the network before it reaches the neighborhood of equilibrium.

4. ***Error Critics*** (Figure 3.6). This provides a general approximation to BTT which is suitable for use in real-time learning [9].

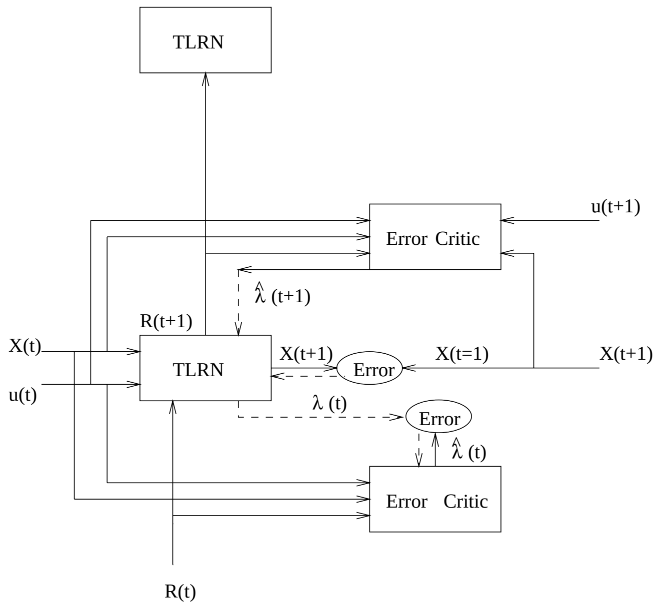

Figure 3.6: Error Critics

5. ***Forward propagation***. This, like BTT, calculates exact derivatives. It is often considered suitable for real-time learning because the calculations go forward in time. However, when there are n neurons and m connections, the cost of this method per unit of time is proportional to $n * m$. Because of this high cost, forward propagation is not really brain-like any more than BTT.

3.3.1 Backpropagation through time(BTT)

BTT is a general method for calculating all the derivatives of any outcome or result of a process which involves repeated calls to the same network or networks used to help calculate some kind of final outcome variable or result E. In some applications, E could represent utility, performance, cost or other such variables. But in this thesis, E will be used to represent error. BTT was first proposed and implemented in [17]. The general form of BTT is as follows:

1)for $k = 1$ to T do forward_calculation(k);

2) calculate result E;

3) calculate direct derivatives of E with respect to outputs of forward calculation;

4) for $k = T$ to 1 backpropagate through forward_calculation(k), calculating running totals where appropriate.

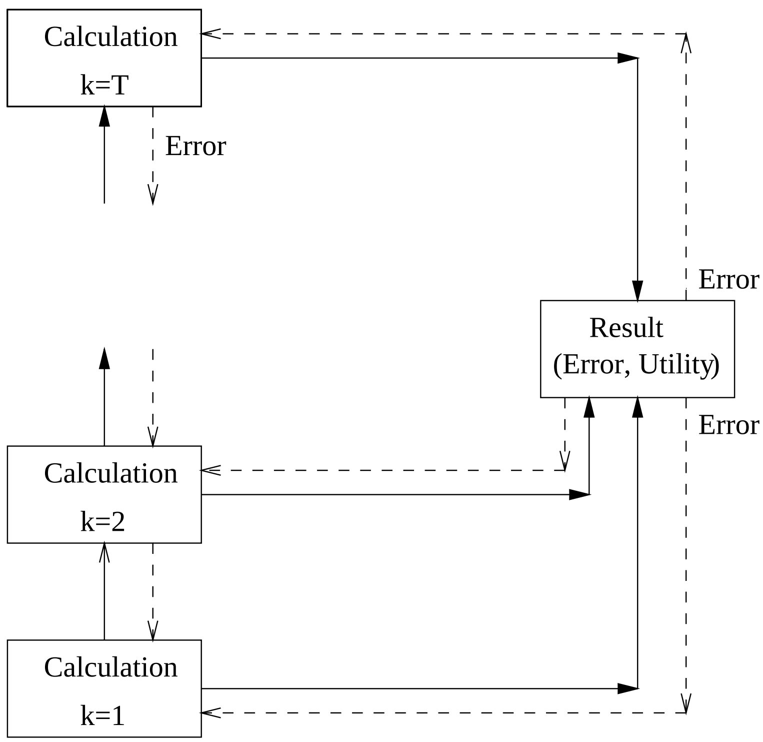

Figure 3.7: Backpropagation through time(BTT)

These steps are illustrated in Figure 3.7. Notice that this algorithm can be applied to all kinds of calculations. Thus we can apply it to cases where k represents data frames t as in the TLRNs, or to cases where k represents internal iterations n as in the SRNs. Also note that each box of calculation receives input from some dashed lines which represent the derivatives of E with respect to the output of the box. In order to calculate the derivatives coming out of each calculation box, one simply uses backpropagation through the calculation of that box starting out from the incoming derivatives. We will explain in more detail how this works in the SRN case and the TLRN case.

So far as we know BTT has been applied in published working systems for TLRNs and for control, but not yet for SRNs until now. However, Rumelhart, Hinton and Williams [18] did suggest that someone should try this.

The application of BTT for TLRNs is described at length in [2] and [9]. The procedure is illustrated in Figure 3.8. In this example the total error is actually the sum of the errors over each time t where t goes from 1 to T. Therefore the outputs of the TLRN at each time t ($t < T$) have two ways of changing total errors:

(1)A direct way when the current predictions $\hat{Y}(t)$ are different from the current targets $Y(t)$;

(2)An indirect way based on the impact of $R(t)$ on errors in later time periods.

Therefore the derivative feedback coming into the TLRN is actually the sum of two feedbacks from two different sources. As a technical detail, note that $R(0)$ needs to be specified somehow. However, we will not discuss this point here because the focus of this thesis is on SRNs.

Figure 3.9 shows the application of BTT to training an SRN. This figure also provides some explanation of our computer code in the appendix. In this figure, the left-hand side (the solid arrows) represents the neural network which predicts our desired output Y. (In our example, Y represents the true values of the J function across all points in the maze). Each box on the left represents a call to a feed-forward system. The vector $X(t)$ represents the external inputs to

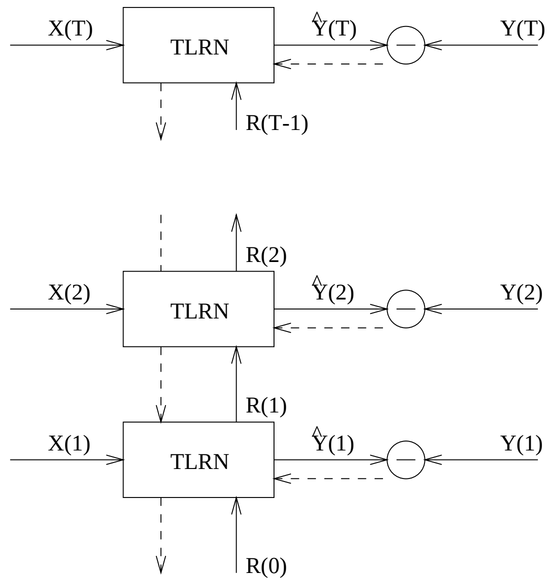

Figure 3.8: BTT for TLRN

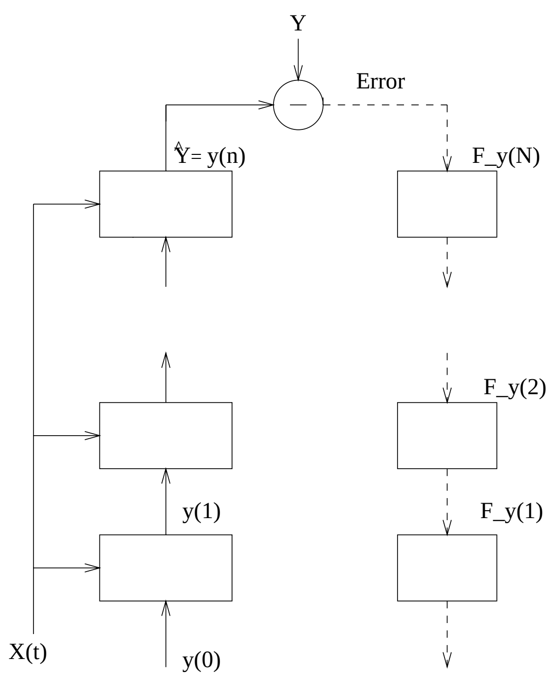

Figure 3.9: BTT for SRN

the entire system. In our case, $X(t)$ consists of two variables, indicating which squares in the maze contain obstacles and which contains the goal respectively. For simplicity, we selected the initial vector $y(0)$ as a constant vector as we will describe below. Each call to the feed-forward system includes calls to a subroutine which implements the generalized MLP.

On the right-hand side of Figure 3.9, we illustrate the backpropagation calculation used to calculate the derivatives. For the SRN, unlike the TLRN, the final error depends directly only on the output of the last iteration. Therefore the last iteration receives feedback only from the final error but the other iterations receive feedback only from the iterations just after them. Each box on the right-hand side represents a backpropagation calculation through the feed-forward system on its left. The actual backpropagation calculation involves multiple calls to the dual subroutine F_net2, which is similar to a subroutine in Chapter 8 of [2].

Notice that the derivative calculation here costs about the same amount as the forward calculation on the left-hand side. Thus BTT is very inexpensive in terms of computer time. However, the backpropagation calculations do require the storage of many intermediate results. Also we know that the human brain does not perform such extended calculations backward through time. Therefore BTT is not a plausible model of true brain-like intelligence. We use it here because it is exact and therefore has the best chance to solve this difficult problem (which was never solved before). In future research, we may try to see whether this problem can also be solved in a more brain-like fashion.

3.3.2 Truncation for SRN

Truncation is probably the most popular method to train SRNs even though the term truncation is not often used. For example, the "simple recurrent networks" used in psychology are typically just SRNs adapted by truncation [19].

Strictly speaking there are two kinds of truncation — ordinary one-step truncation (Figure 3.10) and multi-step truncation which is actually a form of BTT. Ordinary truncation is by far the most popular. In the derivative calculation of ordinary truncation, the memory inputs to the last iteration are treated as if they were fixed external inputs to the network. In truncation there is only one pass of ordinary backpropagation involving only the last iteration of the network. Many people have adapted recurrent networks in this simple way because it seems so obvious. However, the derivatives calculated in this way are not exactly the same because they do not totally represent the impact of changing the weights on the final error. The reason for this is that changing the weights will change the inputs to the final iteration. It is not right to treat these inputs as constants because they are changed when the weights are changed.

The difference between truncation and BTT can be seen even in a simple scalar example, where n=2 and the feed-forward calculation is linear. In this case, the feed-forward calculation is:

$$y(1) = A * y(0) + B * X \tag{3.5}$$

$$y(2) = A * y(1) + B * X \tag{3.6}$$

In addition,

$$Error = E = \frac{1}{2}(Y - y(2))^2 \tag{3.7}$$

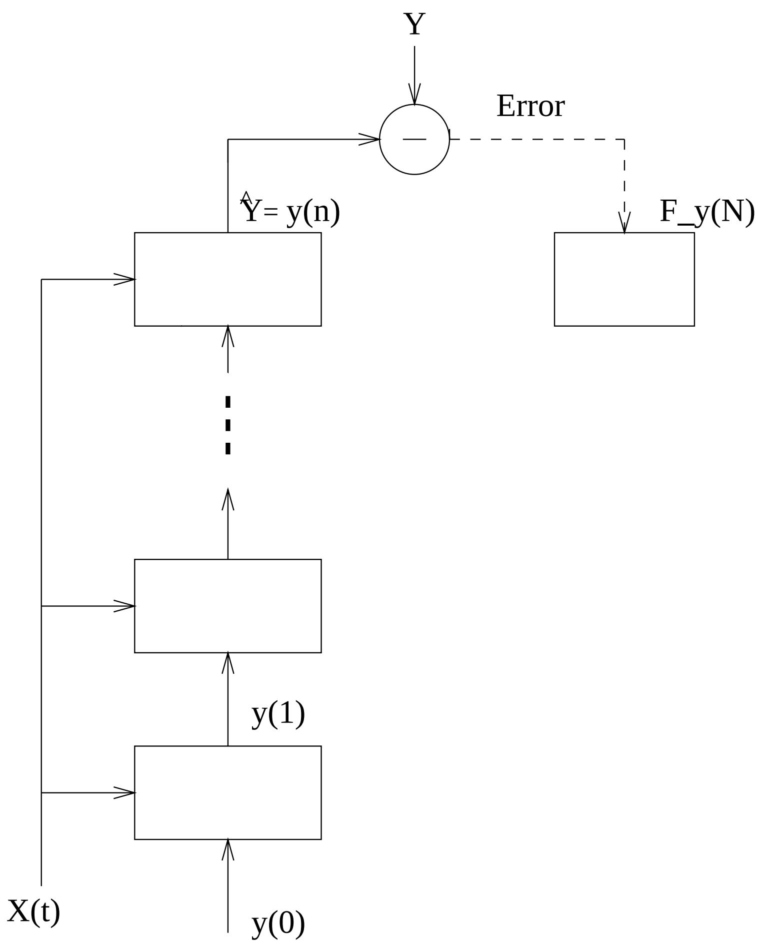

Figure 3.10: Truncation

$$\frac{\partial E}{\partial y(2)} = y(2) - Y \tag{3.8}$$

In truncation, we use Equation (3.6) and deduce:

$$\frac{\partial E}{\partial B} = \frac{\partial E}{\partial y(2)} * \frac{\partial y(2)}{\partial B} = (y(2) - Y) * X \tag{3.9}$$

But for an exact calculation, we substitute (3.5) into (3.6), deriving:

$$y(2) = A^2 * y(0) + A * B * X + B * X \tag{3.10}$$

which yields:

$$\frac{\partial E}{\partial B} = (y(2) - Y) * (A * X + X) \tag{3.11}$$

The result of Equation (3.9) is usually different from the result of Equation (3.11), which is the true result, and comes from BTT. Depending on the value of A, these results could even have opposite signs.

In this thesis, we have used truncation because it is so easy and so popular. If truncation had worked, it would be the easiest way to solve this problem. However, it did not work.

3.3.3 Simultaneous Backpropagation

Simultaneous backpropagation is a method developed independently in different forms by Werbos, Almeida and Pineda [20][21][22]. The most general form of this method for SRNs can be found in chapter 3 of [9] and in [23]. This method is guaranteed to converge to the exact derivatives for the neighborhood of the equilibrium $y(\infty)$ in the case where the forward calculations have reached equilibrium [20].

As with BTT, the derivative calculations are not expensive. Unlike BTT there is no need for intermediate storage or for calculation backwards through time. Therefore simultaneous backpropagation could be plausible as a model of learning in the brain. On the other hand, these derivative calculations do not account for the details of what happened in the early iterations. Unlike BTT, they are not guaranteed to be exact in the case where the final $y(n)$ is not an exact equilibrium. Even in modeling the brain there may be some need to train SRNs so as to improve the calculation in early iterations. In summary, though simultaneous backpropagation may be powerful enough to solve this problem, there was sufficient doubt that we decided to wait until later before experimenting with this method.

3.3.4 Error Critic

The Error Critic, like simultaneous backpropagation, provides approximate derivatives. Unlike simultaneous backpropagation, it has no guarantee of yielding exact results in equilibrium. On the other hand, because it approximates BTT directly in a statistically consistent manner, it can account for the early iterations. Chapter 13 of [9] has argued that the Error Critic is the only plausible model for how the human brain adapts the TLRNs in the neocortex. It would be straightforward in principle to apply the Error Critic to training SRNs as well.

Figure 3.6 shows the idea of an Error Critic for TLRNs. This figure should be compared with Figure 3.9. The dashed input coming into the TLRN in Figure 3.6 is intended to be an approximation of the same dashed line coming into the TLRN in the Figure 3.8. In effect, the Error Critic is simply a neural network trained to approximate the complex calculations which lead up to that dashed line in the Figure 3.7. The line which ends as the dashed line in Figure 3.6 begins as a solid line because those derivatives are estimated as the ordinary output of a neural network, the Error Critic. In order to train the Error Critic to output such approximations, we use the error calculation illustrated on the lower right of Figure 3.6. In this case, the output of the Error Critic from the previous time period is compared against a set of targets coming from the TLRN. These targets are simply the derivatives which come out of the TLRN after one pass of backpropagation starting from these estimated derivatives from the later time period. This kind of training may seem a bit circular but in fact it has an exact parallel to the kind of bootstrapping used in the well known designs for adaptive critics or ADP.

As with simultaneous backpropagation, we intend to explore this kind of design in the future, now that we have shown how SRNs can in fact solve the maze problem.

3.3.5 Forward Propagation

The major characteristics of this method have been described above. This method has been independently rediscovered many times with minor variations. For example, in 1981 Werbos called it conventional perturbation [2]. Williams has called it the Williams – Zipser method [5]. Narendra has called it dynamic backpropagation.

Nevertheless, because this method is more expensive than BTT, has no performance advantage over BTT, and is not plausible as a model of learning in the brain, we see no reason to use it.

CHAPTER 4

Three Test Problems and Details on Architecture and Learning Procedures

In this thesis we use three examples to show that the SRN design has more general function approximation capabilities than does the MLP. Our primary focus was on the maze problem because of its relation to intelligent control as discussed in Chapters 1 and 2. However, before studying this more specialized example, we performed a few experiments on a more general problem which we call Net A/Net B. In this chapter we discuss these two problems in more detail. In addition, in this chapter we describe the two special features – cellular architecture and adaptive learning rate (ALR) used for the maze problem.

4.1 Net A/Net B

In the Net A/Net B problem, our fundamental goal is to explore the idea that the functions that an MLP can approximate are a subset of those that an SRN can. In other words, we hypothesize that an SRN can learn to approximate any functions which an MLP can represent without adding too much complexity, but not vice versa. To consider the functions which an MLP can represent, we can simply sample a set of randomly selected MLPs, and then test the ability of SRNs to learn those functions. Similarly we can generate SRNs at random and test the ability of MLPs to learn to approximate the SRNs.

In order to implement this idea, we used the approach shown in Figure 4.1. The first step in the process was to pick Net A at random. In some experiments, Net A was an SRN, while in other experiments, it was an MLP. In all these experiments, Net B was chosen to be the opposite kind of network than Net A. In picking Net A, we always used the same feed-forward structure. But we used a random number generator to set the weights. After Net A was chosen and Net B was initialized, we generated a stream of random inputs between -1 and +1 following a uniform distribution. For each set of inputs, we trained Net B to try to imitate the output of Net A. Of course Net A was fixed. The results gave an indication of the ability of Net B to approximate Net A.

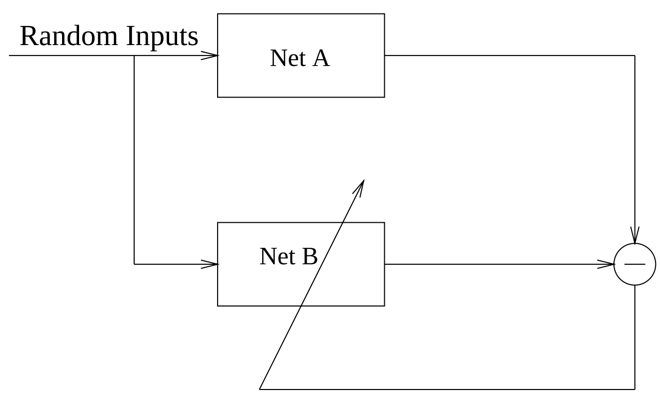

Figure 4.1: Net A/Net B

Our preliminary experiments did show that the SRNs have some advantage over the MLPs. However, in all of these experiments, the SRN was trained with truncation, not BTT. To fully explore all the theoretical issues would require a much larger set of computer runs. Still, these initial experiments were very useful in testing some general computer codes in order to prepare for the complexities of the maze problem.

4.2 The Maze Problem

In the classic form of the maze problem, a little robot is asked to find the shortest path from the starting position to a goal position on a two-dimensional surface where there are some obstacles. For simplicity, this surface is usually represented as a kind of chess board, or grid of squares, in which every square is either clear or blocked by an obstacle. In formal terms, this means that we can describe the state of the maze by providing three pieces of information:

(1) An array $\boldsymbol{A}$ which has the value 0 when the square is clear and 1 when it is covered by an obstacle;

(2) The coordinates of the goal;

(3) The coordinates of the starting square.

In our case, we used a large number to represent the obstacles.

As discussed in Chapter 1, many researchers have trained neural networks to learn an individual maze [5]. Our goal was to train a network to input the array $\boldsymbol{A}$ and to output the array $\boldsymbol{J}$ for all the clear squares. According to dynamic programming, the best strategy of motion for a robot is simply to move to that neighboring square which has the smallest $\boldsymbol{J}$.

This more general problem has not been solved before with neural networks. For example, Houillon etc. [24] initially attempted to solve this problem with MLPs, but were unsuccessful. Widrow in several plenary talks has reported that his neural truck backer upper has some ability to see and avoid obstacles. However, this ability was based on an externally developed potential function which was not itself learned by neural networks. Such potential functions are analogous to the $\boldsymbol{J}$ function which we are trying to learn.

In fact, this maze problem can always be solved directly and economically by dynamic programming. Why then do we bother to use a neural network on this problem? The reason for using this test is not because this simple maze is important for its own sake, but because this is a very difficult problem for a learning system, and because the availability of the correct solution is very useful for testing. It is one thing for a human being to know the answer to a problem. It is a different thing to build a learning system which can figure out the answer for itself. Once the simple maze problem is fully conquered, we can then move on to solve more difficult navigation problems which are too complex for exact dynamic programming.

In order to represent the maze problem as a problem for supervised learning, we need to generate both the inputs to the network (the array $\boldsymbol{A}$) and the desired outputs (the array $\boldsymbol{B}$) (Refer to the Appendix). For this basic experiment, we chose to study the example maze shown in Figure 4.2. In this figure, G represents the goal position, which is assigned a value of "1"; the other numbers represent the true values of the $\boldsymbol{J}$ function as calculated by dynamic programming (subroutine "Synthesis" in the attached code in the Appendix). Intuitively each $\boldsymbol{J}$ value represents the length of the shortest path from that square to the goal.

Initially we chose to study this particular maze because it poses some very unique difficulties. In particular there are four equally good directions starting

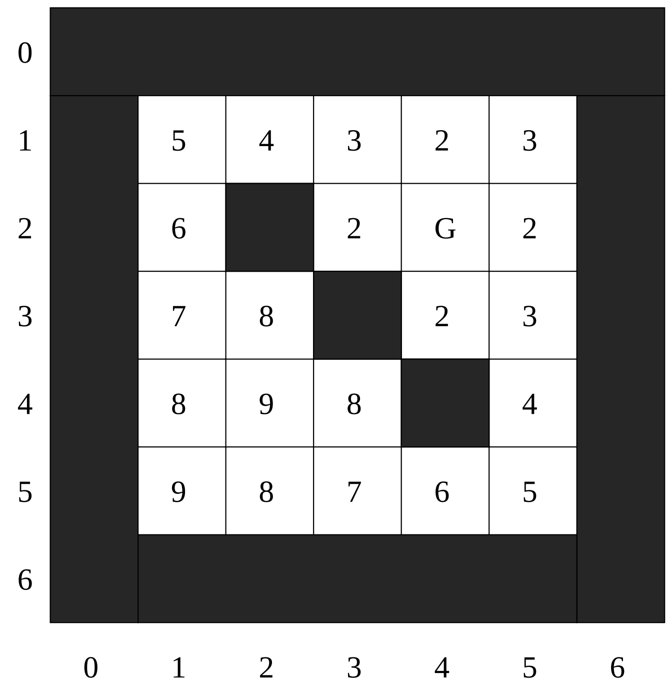

Figure 4.2: Desired J function of a maze

from one of these squares in this maze — a feature which can be very confusing to neural networks, even human. If we had used a fully connected conventional neural network, then the use of a single test maze would have led to over-training and meaningless results. However, as we will discuss later in this chapter, we constrained all of our networks to prevent this problem. Nevertheless, a major goal of our future research will be to test the ability of SRNs to predict new mazes after training on different mazes.

This problem of maze navigation has some similarity to the problem of connectedness described by Minsky [25]. Logically we know that the desired output in any square can depend on the situation in any other square. Therefore, it is hard to believe that a simple feed-forward calculation can solve this kind of problem. On the other hand, the Bellman equation (Equation(2.1)) itself is a simple recurrent equation based on relationships between “neighboring” (successive) states. Therefore it is natural to expect that a recurrent structure could approximate a $\boldsymbol{J}$ function. The empirical results in this thesis confirm these expectations.

4.3 Details for the Net A/Net B Problem

In all these experiments, the MLP network and the feed-forward network f in the SRN was a standard MLP with two hidden layers. The input vector X consisted of six numbers between -1 and +1. The two hidden layers and the output layers all had three neurons. The initial weights were chosen at random according to a uniform distribution between -1 and +1. Training was done by standard backpropagation with a learning rate of 0.1.

4.4 Weight-sharing and Cellular Architecture

4.4.1 What is Weight-sharing?

In theoretical terms, weight-sharing is a generalized technique for exploiting prior knowledge about some symmetry in the function to be approximated. Weight-sharing has sometimes been called "windowing" or "Lie Group" techniques.

Weight-sharing has been used almost exclusively for applications like character recognition or image processing where the inputs form a two-dimensional array of pixels [3][26]. In our maze problem the inputs and outputs also form arrays of pixels. Weight-sharing leads to a reduction in the number of weights. Fewer weights lead in turn to better generalization and easier learning.

As an example, suppose that we have an array of hidden neurons with voltages $net[ix][iy]$, while the input pixels form an array $X[ix][iy]$. In that case, the voltages for a conventional MLP would be determined by the equation:

$$net[i][j] = \sum_{ix,iy} W(i,j,ix,iy) * X(ix,iy) \tag{4.1}$$

Thus if each array has a size $20 * 20$, the weights form an array of size $20 * 20 * 20 * 20$. This means 160,000 weights — a very big problem. In basic weight-sharing, this equation would be replaced by:

$$net[i][j] = \sum_{d1,d2} W(d1,d2) * X(i+d1, j+d2) \tag{4.2}$$

Furthermore, if $d1$ and $d2$ are limited to a range like $[-5,5]$, then the number of weights can be reduced to just over 100. Actually this would make it possible to add two or three additional types of hidden neurons without exceeding 1,000 weights. This trick was used by Guyon etc. [3]. They used it to develop the most successful zip code digit recognizer in existence.

Intuitively $AT\&T$ justified this idea by arguing that similar patterns in different locations have similar meanings. However, there is a more rigorous mathematical justification for this procedure as we will see.

4.4.2 Lie Group Symmetry and Weight-sharing

The technique of weight-sharing in neural networks is really just a special case of the Lie-group method pioneered much earlier by Laveen Kanal [42] in image processing. Formally speaking, if we know that the function F to be approximated must obey a certain symmetry requirement then we can impose the same symmetry on the neural network which we use to approximate F. More precisely, if $Y = F(x)$ always implies that $MY = F(Mx)$, where M is some kind of simple linear transformation, then we can require that the neural network possess the same symmetry.

Both in image processing and in the maze problem, we can use the symmetry with respect to those transformations M which move all the pixels by the same distance to the left, to the right or up and down. In the language of physics, these are called spatial translations.

Because we know that the best form of the neural network must also obey this symmetry, we have nothing to lose by restricting our weights as required by the symmetry.

4.4.3 How We implemented Weight-sharing

In order to exploit Lie group symmetry in a rigorous way, we first reformulated the task to be solved so as to ensure exact Lie group symmetry. To do this, we designed our neural network to solve the problem of maze defined over a torus. For our purposes, a torus was simply an N by N square where the right-hand neighbor of $[i, N]$ is the point $[i, 0]$, and likewise for the other edges. This system can still solve an ordinary maze as in Figure 4.2, where the maze is surrounded by walls of obstacles.

Next we used a cellular structure for our neural network including both the MLPs and SRNs. A cellular structure means that the network is made up of a set of cells each made up of a set of neurons. There is one cell for each square in the maze. The neurons and the weights in each cell are the same as those in any other cell. Only the inputs and outputs are different because they come from different locations.

The general idea of our design is shown in Figure 4.3. Notice that each cell is made up of two parts: a connector part and a local memory part which includes 4 neighbors and the memory from itself in Figure 4.3. The connector part receives the inputs to the cell and transmits its output to all four neighboring cells. In addition, the local memory part sends all its outputs back as inputs, but only to the same cell. Finally the forecast of $\boldsymbol{J}$ is based on the output of the local memory part.

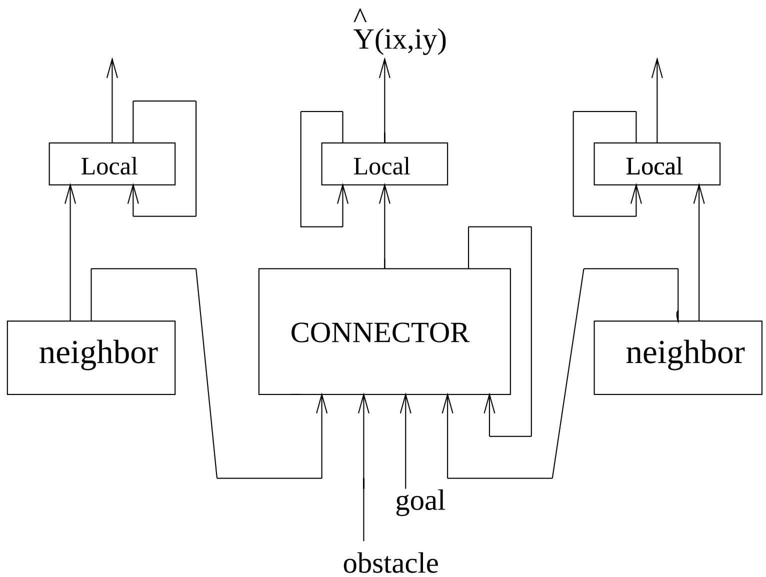

Figure 4.3: General Idea of the Cellular Network

The exact structure which we used is shown completely in Figure 4.4. In this figure it can be seen that each cell receives 11 inputs on each iteration. Two of these inputs represent the goal and obstacle variables for the current pixel. The next four inputs represent the outputs of the connector neuron from the four neighboring cells from the previous iteration. The final five inputs are simply the outputs of the same cell from the previous iteration. Then after the inputs,

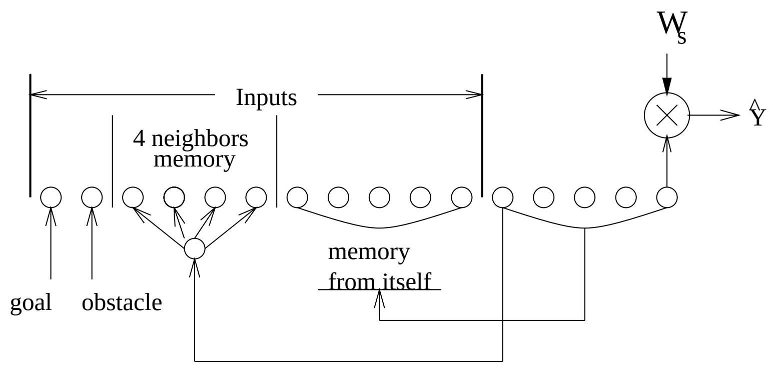

Figure 4.4: Inputs, Outputs and Memory of Each Cell

there are only five actual neurons. The connector part is only one neuron in our case. The local memory part is four neurons. The prediction of $J[ix][iy]$ results from multiplying the output of the last neuron by W_s, a weight used to rescale the output.

To complete this description, we must specify how the five active neurons work. In this case, each neuron takes inputs from all of the neurons to its left, as in the generalized MLP design [2]. Except for $\hat{J}$, all of the inputs and outputs range between -1 and 1, and the tanh function is used in place of the usual sigmoid function.

To initialize the SRN on iteration zero, we simply picked a reasonable looking constant vector for the first four neurons out of the five. We set the initial starting value to -1. For the last neuron, we set it to 0. In future work, we shall probably experiment with the adaptation of the starting vector $y(0)$.

In order to backpropagate through this entire cellular structure, we simply applied the chain rule for ordered derivatives as described in [2].

4.5 Adaptive Learning Rate

In our initial experiments with this structure, we used ordinary dynamic programming with only one special trick. The trick was that we set the number of iterations for SRN to only 1 on the first 20 trials, and then to 2 for the next 20 trials... and so on up until there were 20 iterations.

We found that ordinary weight adjustment led to extremely slow learning due to oscillation. This was not totally unexpected because slow learning and oscillation are a common result of simple steepest descent methods. There are many methods available to accelerate the learning. Some of these like the DEKF method developed by Ford Motor Company are similar to quasi-Newton methods [27] which are very powerful but also somewhat expensive. For this work we chose to use a method called the adaptive learning rate (ALR) as described in chapter 3 of [9]. This method is relatively simple and cheap, but far more flexible and powerful than other simple alternatives.

In this method, we maintain a single adapted learning rate for each group of weights. In this case, we chose three groups of weights (refer to the Appendix):

1. The weight W_s used for rescaling of the output;
2. The constant or bias weights ww;
3. All the other weights W.

For each group of weights the learning rate is updated on each trial according to the following formula:

$$LR(t+1) = LR(t) * (0.9 + 0.2 * \frac{\sum_k W_k(t) * W_k(t-1)}{\sum_k W_k(t-1) * W_k(t-1)}) \qquad (4.3)$$

where the sum over k actually refers to the sum over all weights in the same group. In addition, to prevent overshoot, we would reset the learning rate to:

$$\frac{LR * E}{\sum_k \left(\frac{\partial E}{\partial W_k}\right)^2} \tag{4.4}$$

where the sum is taken over all weights. In this special case where the error on the next iteration would be predicted to be less than zero, i.e.:

$$\begin{aligned}
& E - \sum_k (W_k(t+1) - W_k(t)) * \frac{\partial E}{\partial W_k}(t) \\
= \; & E - \sum_k (LR * \frac{\partial E}{\partial W_k}(t)) * \frac{\partial E}{\partial W_k}(t) \\
= \; & E - LR * \sum_k \left(\frac{\partial E}{\partial W_k}(t)\right)^2 < 0
\end{aligned} \tag{4.5}$$

where $W_k(t+1)$ is the new value for the weights which would be used if the learning rates were not reset. In our case, we modified this procedure slightly to apply it separately to each group of weights.

After the adaptive learning rates were installed the process of learning became far more reliable. Nevertheless, because of the complex nature of the function $\boldsymbol{J}$, there was still some degree of local minimum problem. For our purposes, it was good enough to simply try out a handful of initial values which we guessed at random. However, in future research, we would like to explore the concept of shaping as described in [9].

4.6 An SRN receiver for OMD

Code division multiple access (CDMA) has recently emerged as an access protocol ideally suited for voice and data transmission. CDMA provides numerous advantages for a range of multiuser applications, including cellular mobile radio networks, personal communication systems, and indoor wireless communications. The only significant limitation of the conventional CDMA system is the "near-far problem" where strong signals interfere with the detection of a weak signal, i.e. when the detection received power of the interfering signals becomes large, severe performance degradation of the system is observed.

Optimum multiuser detectors assume knowledge of all the modulation waveform and channel parameters, and exploit this information to eliminate multiple-access interference and to achieve near-far resistance. However, the computational complexity of the optimal multiuser detector (OMD) grows exponentially with the number of users. Since the optimum multiuser demodulation is an NP-complete problem, research efforts have concentrated on the development of suboptimal receivers.

Neural networks have shown their abilities of nonlinear functional approximation, adaptive learning and parallel computing. So far, mainly two kinds of neural networks, multi-layer perceptrons (MLPs) and Hopfield neural networks (HNNs) have been applied in multiuser detection to construct suboptimal multiuser receivers.

Hopfield neural networks (HNNs) have the abilities to solve static optimization problems. There have been some research on their implementation of OMD [33][34] because minimizing the objective function of the OMD can be translated into minimizing an HNN "energy" function. However, all the weights and connections must be specified in advance when the HNNs are used for static optimization. This limits the success of HNNs' receiver. So, when we do not have the information of the matrix of the signal cross-correlations, we have to use other kinds of neural networks with weight adapting abilities.

Multi-layer perceptrons(MLPs) trained with backpropagation have been used to approximate the optimum function [35]. It has been shown that MLPs have the capabilities of approximating arbitrary decision regions in the input space for most classification problems. However, since the SRNs have better functional approximate abilities than the MLPs, as we have shown in the NetA/NetB and maze problems, we tried to use the SRNs to solve this problem.

4.6.1 Multiuser Detection Schemes

We consider a K-user binary communication system, employing normalized modulation waveforms $s_1, s_2, \cdots, s_k$, and signaling through an additive white Gaussian noise channel. The received signal in such a channel can be modeled as

$$r(t) = S(t) + \sigma\eta(t) \tag{4.6}$$

where $\eta(t)$ is white Gaussian noise with unit power spectral density and $S(t)$ is the superposition of the data signals of the K users, given by

$$S(t) = \sum_{k=1}^{K} A_k \sum_{i=-M}^{M} b_k(i) s_k(t - iT - \tau_k) \tag{4.7}$$

where A_k is the received amplitude of the kth user; $(2M+1)$ is the frame length; $b_k(i)$ is the ith symbol of the kth user (assumed to be binary, ± 1); τ_k is the relative delay of the kth user; and T is the inverse of the data rate.

It is assumed that the s_k, the kth user's normalized signaling waveform, is supported only on the interval [0,T]; and that $\{b_k(i)\}$ is a collection of independent equiprobable ± 1 random variables.

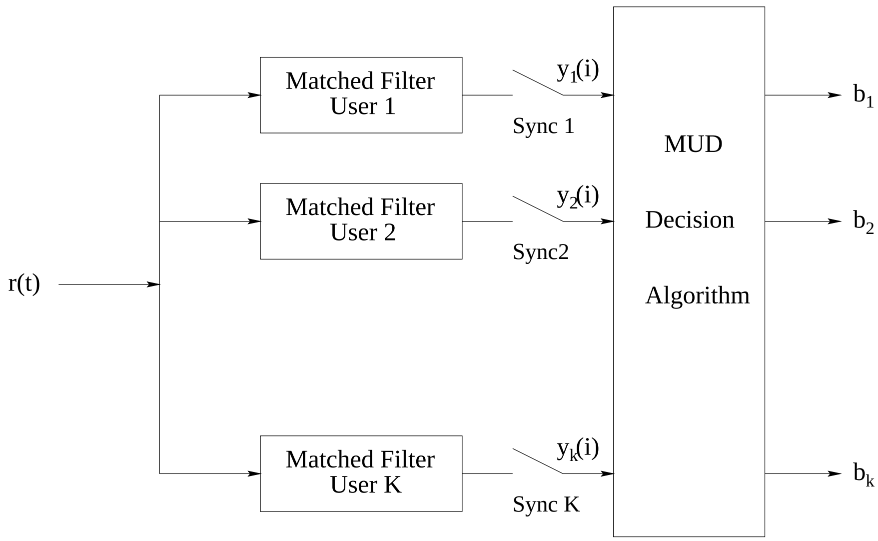

Figure 4.5: The general structure of multiuser detection(MUD)

Here we consider the synchronous case of the model (4.7), in which $\tau_1 = \tau_2 = \cdots = \tau_K \equiv 0$. Since the full model (4.7) can be viewed as a synchronous model with $(2M+1)K$ users [32], this restriction is not significant for the purposes of bit-error-rate analysis. In this synchronous case, it is easily seen that a sufficient statistic for demodulating the ith data bits of the K users is given by the K-vector $\boldsymbol{y}$ whose kth component is the output of a filter matched to s_k in the ith data interval, i.e.,

$$y_k(i) = \int_{it}^{(i+1)t} s_k(t - iT)r(t)dt, \quad k = 1, 2, \cdots, K. \tag{4.8}$$

Statistically, this problem is invariant to the choice of the symbol interval i, and so, without loss of generality, we consider the case $i = 0$, and suppress the index i.

This sufficient vector $\boldsymbol{y}$ can be written as:

$$\boldsymbol{y} = \boldsymbol{RAb} + \sigma\boldsymbol{\eta} \tag{4.9}$$

where $\boldsymbol{R}$ is the $K \times K$ matrix of signal cross-correlations:

$$r_{k,l} = \int_0^T s_k(t)s_l(t)dt \tag{4.10}$$

$\boldsymbol{A} = \text{diag}\{A_1, A_2, ...A_K\}$; $\boldsymbol{b}$ is a K-vector whose kth component is b_k; and $\boldsymbol{\eta}$ is a $\mathcal{N}(0, \boldsymbol{R})$ random vector, independent of $\boldsymbol{b}$.

So far, several demodulators have been studied for this channel [37], including the conventional detector

$$\hat{b}_k = sgn(y_k), \tag{4.11}$$

and the optimum multiuser detector

$$\hat{\boldsymbol{b}} = arg \quad max_{b\in\{-1,+1\}^K} 2\boldsymbol{y}^{(i)}\boldsymbol{b} - \boldsymbol{b}^T\boldsymbol{Rb} \tag{4.12}$$

Because of the exponential growth of the computational complexity of the OMD with the number of active users, suboptimal detection schemes have been proposed. According to (4.12) and Figure 4.5, the optimum MUD essentially is to decide and estimate each user transmitted signal $\boldsymbol{b}$ from the viewing vector $\boldsymbol{Y}$ by the minimum error probability. So the optimum MUD problem is the problem of decision and classification. One kind of detection scheme is using neural networks, which have been successfully applied to many classification problems and functional approximation problems. Figure 4.6 is the structure of a neural network multiuser receiver for multiuser detection.

4.6.2 Neural net receiver

There are mainly two kinds of neural net receivers that have been used. One is using a multi-layer perceptrons structure trained by back-propagation algorithm; the other is Hopfield neural networks(HNNs). In this thesis, we will introduce a new kind of neural networks – simultaneous recurrent network (SRN) and show that it can do better than MLPs and HNNs as a kind of suboptimal detection scheme.

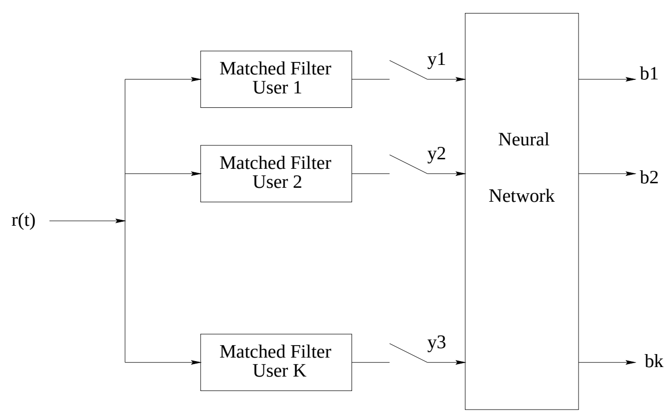

Figure 4.6: A neural network receiver for multiuser demodulation

HNNs are single layer networks with output(s) feedback consisting of simple processors(neurons) that can collectively provide good solutions to difficult optimization problems. An HNN is depicted in Figure 4.7. A connection between two processors in established through a conductance T_{ij} which transforms the voltage outputs of amplifier j to a current input for amplifier i. Externally supplied bias currents I_i are also present in every processor j.

Each neuron i receives a weighted sum of the activations of other neurons in

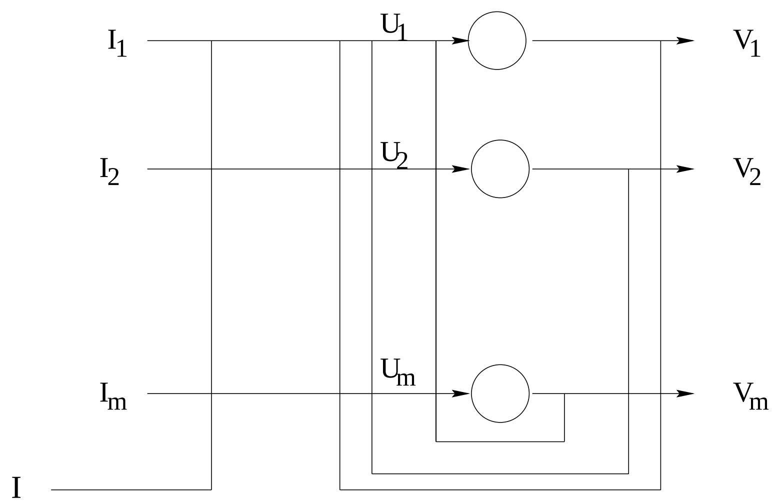

Figure 4.7: Hopfield neural network

the network, and updates its activation according to the rule

$$V_i = g(U_i) = g(\sum_{j \neq i} T_{ij} V_j + I_i) \tag{4.13}$$

The function $g(U_i)$ can be either a binary or antipodal thresholding function for the case of the McCulloch-Pitts neuron

$$V_i = g(U_i) = sign(U_i) \tag{4.14}$$

or any monotonically increasing nonlinear function. One example of such a nonlinear function often used in simulations is the sigmoid function, defined by

$$V_i = g(U_i) = sigm(\alpha U_i) = \frac{1 - e^{-\alpha U_i}}{1 + e^{-\alpha U_i}} \tag{4.15}$$

where α is a positive constant that controls the slope of the nonlinearity. In particular, when $\alpha \to \infty$, then $g(U_i) \to sign(U_i)$.

It has been shown [40] that, in the case of symmetric connections ($T_{ij} = T_{ji}$), the equations of motion for the activation of the neurons of an HNN always lead to convergence to a stable state, in which the output voltages of all the amplifiers remain constant. Also, when the amplifier gain curve is narrow, (i.e., the non-linear activation function $g(\cdot)$ approaches the antipodal thresholding function), the stable states of a network with N neuron units are the local minima of the quantity(energy function)

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} T_{ij}V_iV_j - \sum_{i=1}^{N} V_iI_i. \tag{4.16}$$

The equation of motion for the ith neuron may be described in terms of the energy function (4.15) as follows:

$$\frac{dU_i}{dt} = -\frac{\partial E}{\partial V_i} - \frac{U_i}{\tau} = -\frac{U_i}{\tau} + \sum_{i \neq j} T_{ij}V_j + I_i \tag{4.17}$$

Since the OMD objective function is very similar to an HNN energy function, some work has been done to apply HNNs to solve this difficult combinatorial optimization problem [33][34][36]. The optimum multiuser detection can be clearly related to the HNN by identifying L and E. As a result, the variables correspond as follows:

$$I_i = 2y_i, \qquad T_{ij} = -2h_{ij}, \qquad V_i = \hat{b}_i, \qquad N = K. \tag{4.18}$$

Thus the number of neurons in an HNN receiver is equal to that of the users in the CDMA system.

However, if we do not have the information of the cross-correlation function of the signal set s_1, s_2 .. s_K, we can think of the problem as a function approximation problem, where we try to figure out the solution of the inverse function of f given $\boldsymbol{y}$, and $\boldsymbol{y}$ satisfies the following function:

$$\boldsymbol{y} = f(\boldsymbol{b}) \tag{4.19}$$

In [35], a two-layered perceptron trained by back-propagation was used for 2-user and 3-user problems. Though it was shown that this kind of MLP receiver have better performance than a conventional receiver, for our examples we can see the MLP does not work so well as the SRN does.

4.6.3 An SRN receiver

Here we consider a system of five-users transmitting synchronously, and use the similar SRN structure trained by BTT as that used for the maze problem with 4 active neurons. Adaptive learning rate technique is also used. However, we do not need to worry about the neighbor's information here. So we simply have five inputs with one input containing the value of the receiver passed the user's matched filter and the other four inputs which keep the outputs as memory.

CHAPTER 5

Simulation Results and Conclusions

Later in this chapter, we review some simulation results for the three test problems discussed before. From analyzing the results, we can conclude that compared to the MLPs, the SRNs are more powerful in nonsmooth function approximation. In addition, our new design — the cellular structure — can really solve the maze problem.

5.1 Results for the Net A/Net B Problem

From Figure 5.1 to Figure 5.4 we can see that the SRN using the same three-layered neural network structure (9 inputs, 3 outputs, and 3 neurons for each hidden layer) as the MLP can achieve better simulation result. The SRN not only converged more rapidly than the MLP(Figure 5.1 and Figure 5.2, but also reached a smaller error(Figure 5.3 and Figure 5.4), about $1.25 * 10^{-4}$, while the MLP reached $5 * 10^{-4}$. Thus we can say that, in this typical case, an SRN has better ability to learn an MLP than an MLP to learn an SRN.

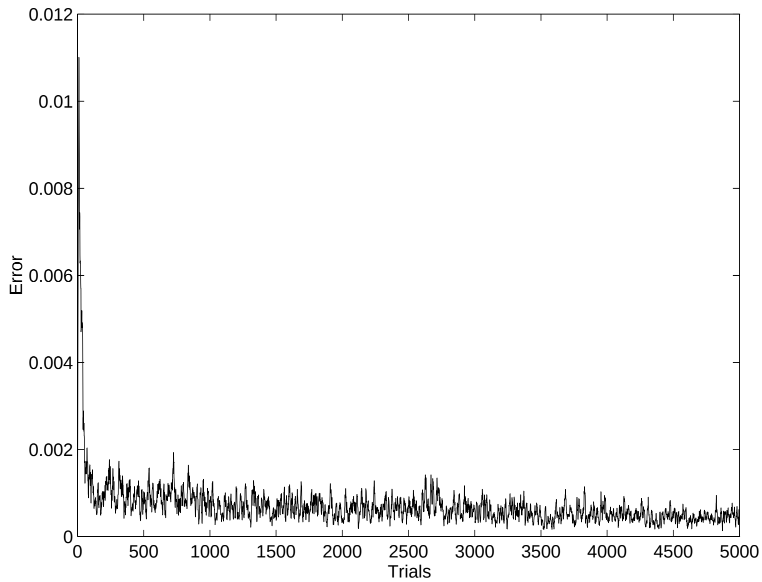

Figure 5.1: The MLP learned the SRN

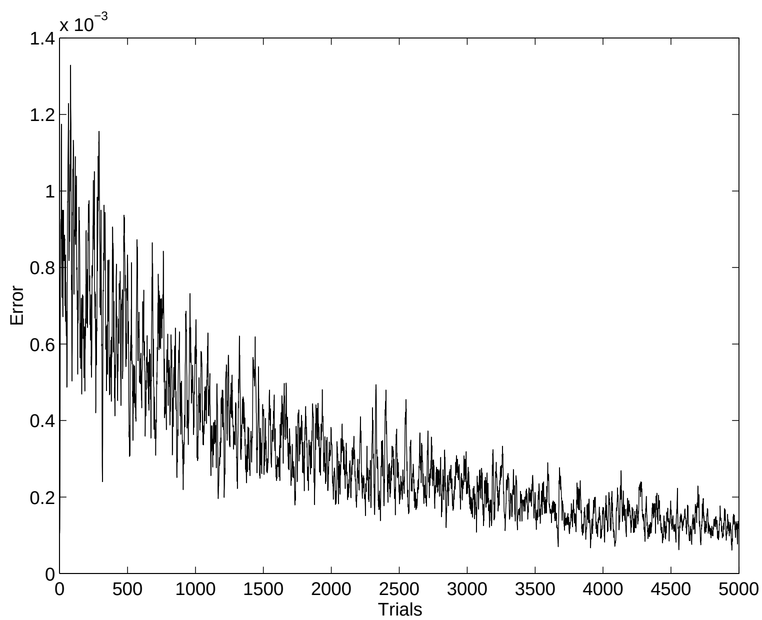

Figure 5.2: The SRN learned the MLP

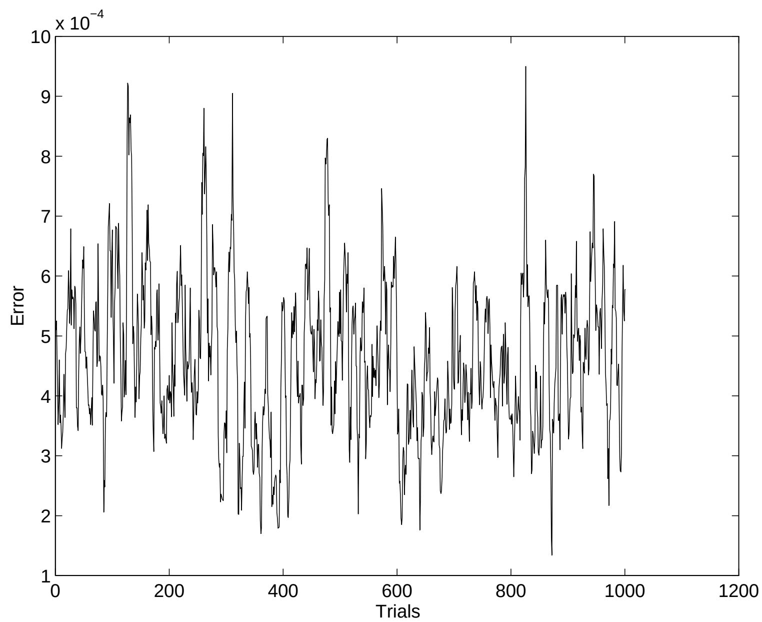

Figure 5.3: The last 1000 trials of Figure 5.1

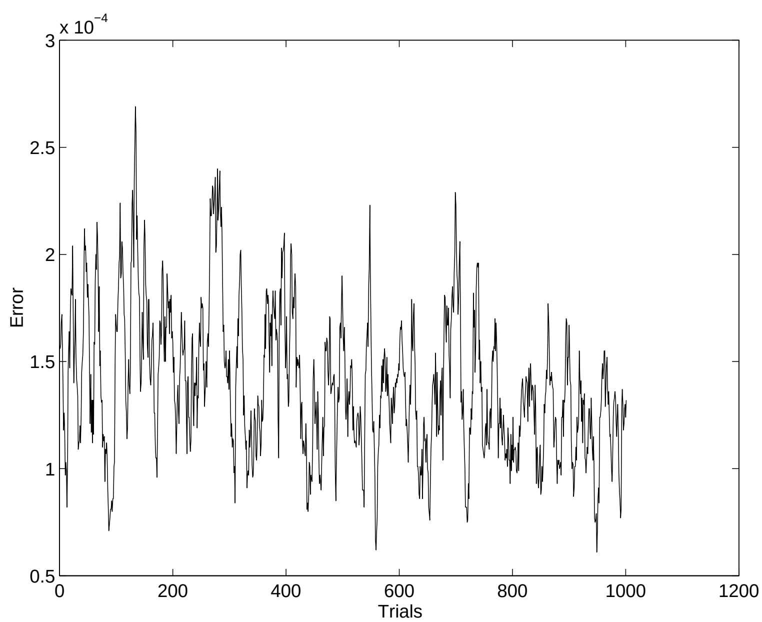

Figure 5.4: The last 1000 trials of Figure 5.2

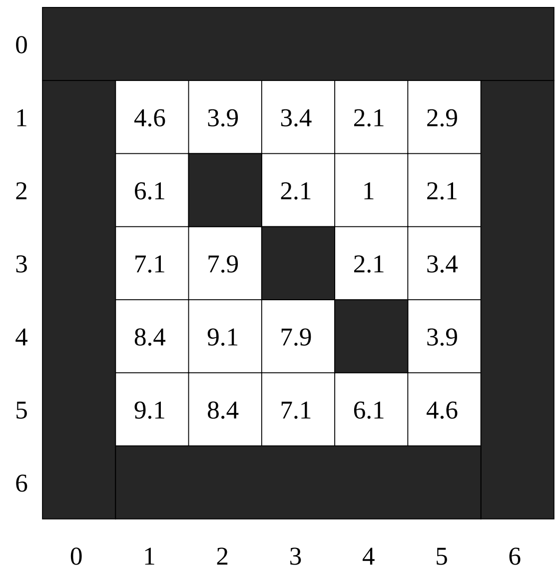

Figure 5.5: J function as predicted by SRN-BTT(I)

5.2 Results for the Maze Problem

There are two parts of the results for the maze problem.

First, we compare the J function in each pixel of the same maze as predicted by an SRN trained by BTT and an SRN trained by truncation respectively with the actual J function for the maze. Figure 5.5 and 5.6 show that the SRN trained by BTT can really approximate the J function, but the SRN trained by truncation cannot, respectively. Moreover, the SRN trained by BTT can learn the ability to find the optimal path from the start to the goal as calculated by dynamic programming. Although there is some error in the approximation of J by the SRN trained by BTT, the errors are small enough that a system governed by the approximation of J would always move in an optimal direction.

Second, we show some error curves from Figure 5.7 to Figure 5.12. From the figures we can see the error curve of SRN trained by BTT not only converged

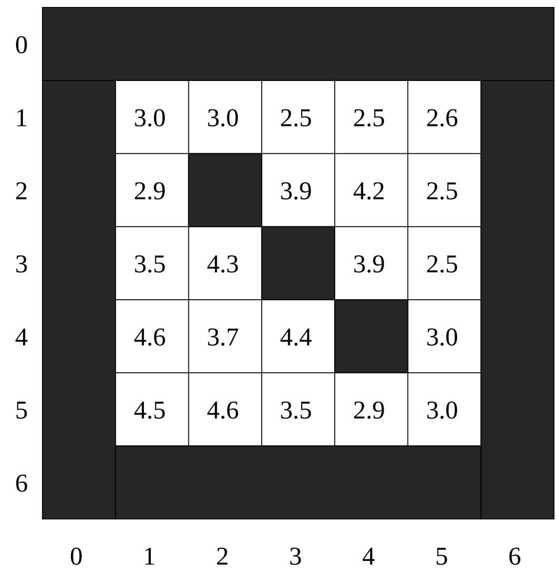

Figure 5.6: J function as predicted by SRN-Truncation(I)

more rapidly than the curve of the SRN trained by truncation, but also reached a much smaller level of error. The errors with the MLP did not improve at all after about 80 trials(Figure 5.11 and Figure 5.12).

5.3 Comparison of the performance of MUD using OMD, SRN and MLP

In Figure 5.13, we compare the MLP, SRN and OMD detectors for 5 active synchronous users by evaluating their bit-error-rate of user 1 via the signal-to-noise ratio (SNR) of user 1. The SNR of the other users were fixed at 8dB. In this example, we trained the SRN 5000 times before doing the test.

In Figure 5.14, the comparison of the BER of the MLP, SRN and OMD detectors for 5 active synchronous users is plotted versus E_i/E_1. In this example, the SNR of the user 1 is fixed at 8dB. The BER of user 1 of the SRN detector

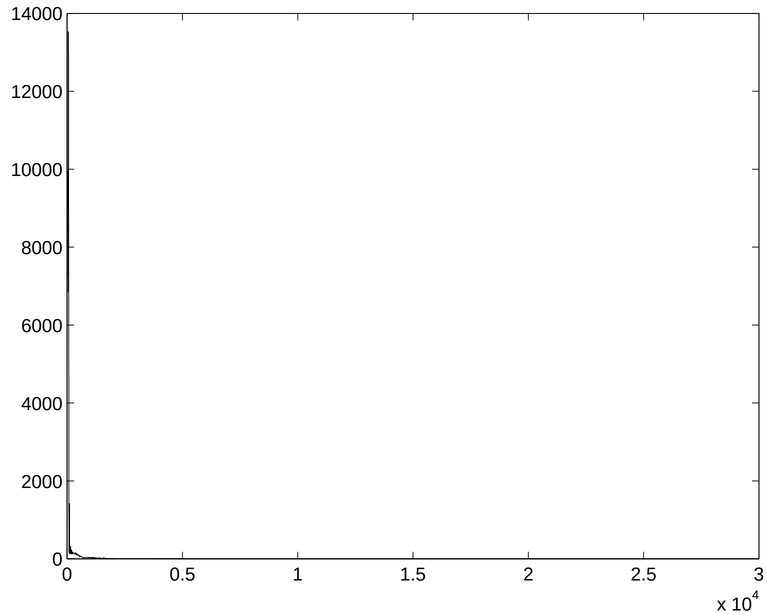

Figure 5.7: Error curve of itJ function as predicted by SRN-BTT(II)

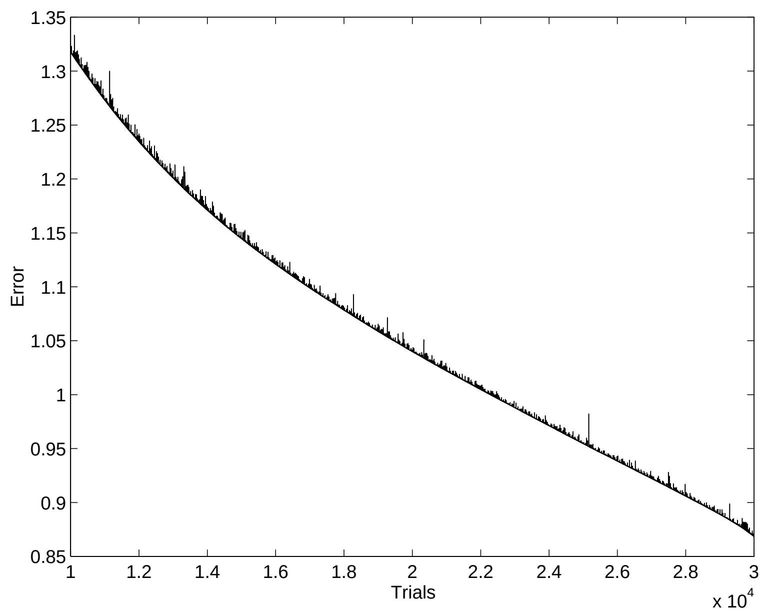

Figure 5.8: Error curve of itJ function as predicted by SRN-BTT(III)

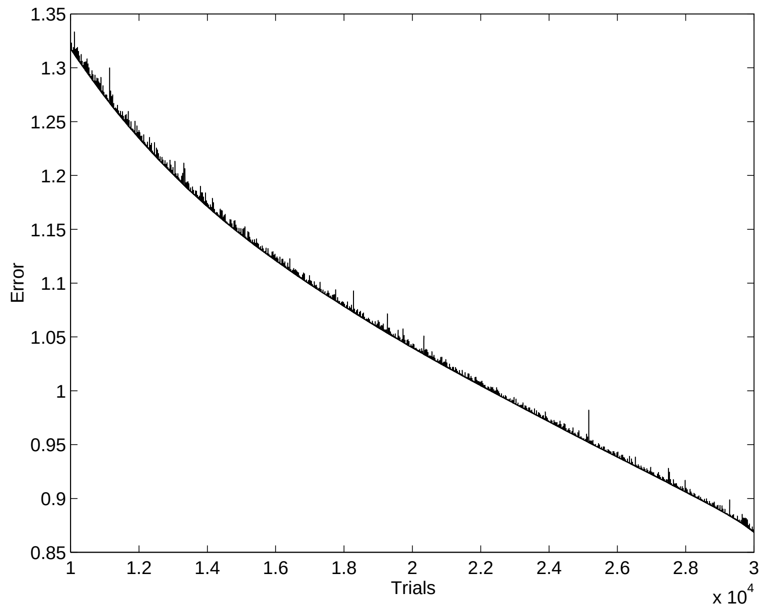

Figure 5.9: Error curve of J function as predicted by SRN-Truncation(II)

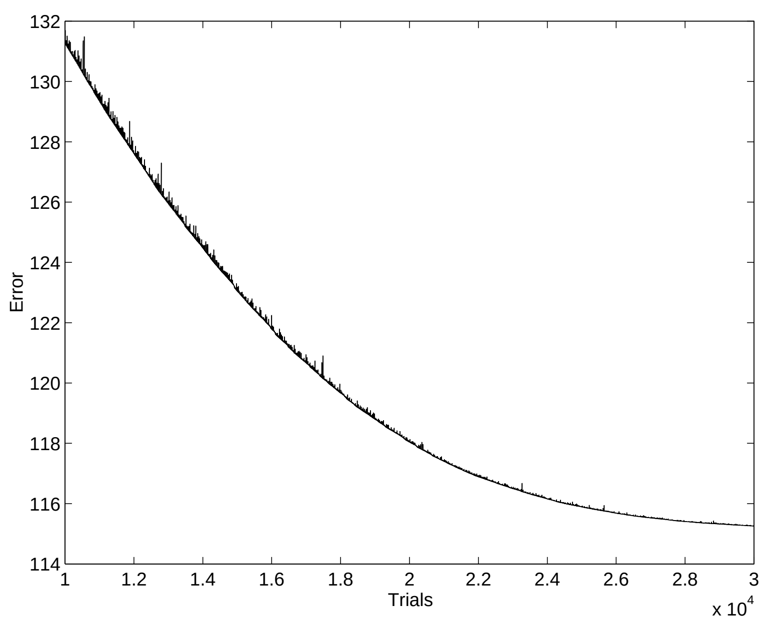

Figure 5.10: Error curve of J function as predicted by SRN-Truncation(III)

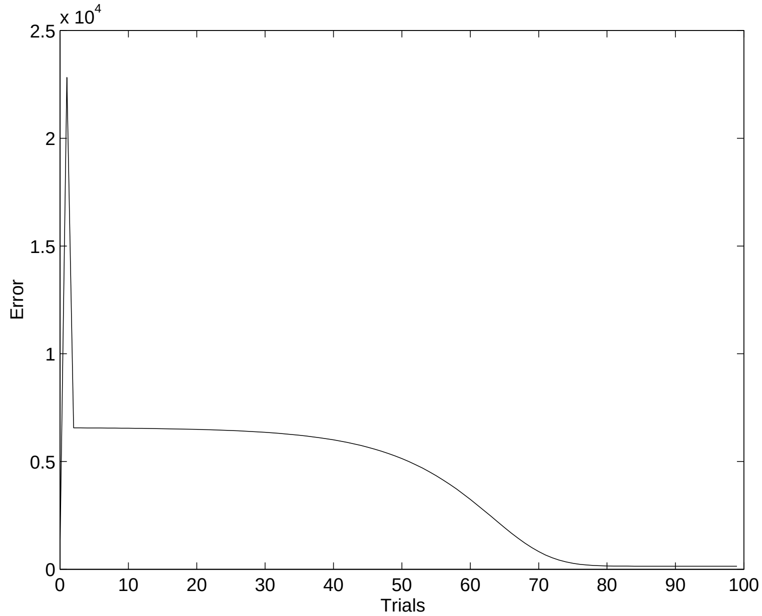

Figure 5.11: Error curve of J function as predicted by MLP(I)

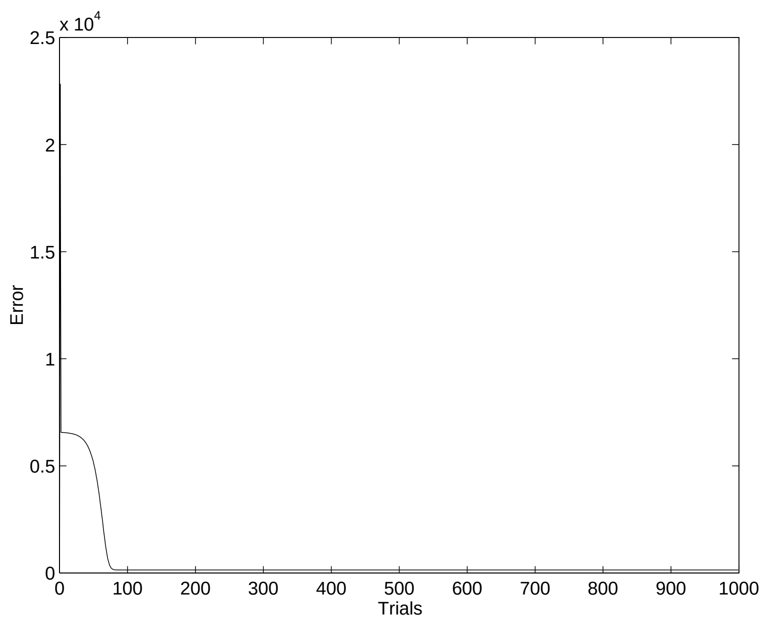

Figure 5.12: Error curve of J function as predicted by MLP(II)

is smaller than that of the MLP detector.

100 tests have been done for the above two experiments, where in each test 100,000 random bits have been tested.

From both these two examples we can see that the SRN exhibits better performance than MLP and can approach that of the OMD.

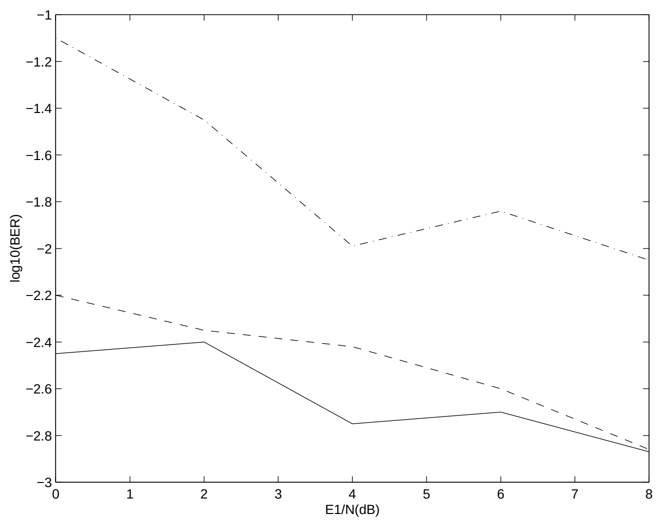

Figure 5.13: Bit error rate versus E_1/N for 5-user channel with the SNR of the other users fixed at 8dB, where solid line is the OMD, the dash line is the SRN and the dash-dot line is the MLP.

5.4 Conclusions

In this thesis, we have described a new neural network design for J function approximation in dynamic programming. We have tested this design in three test problems: Net A/Net B, the maze and the multiuser detection problems. In the Net A/ Net B problem, we showed that SRNs can learn to approximate MLPs better than MLPs can learn SRNs.

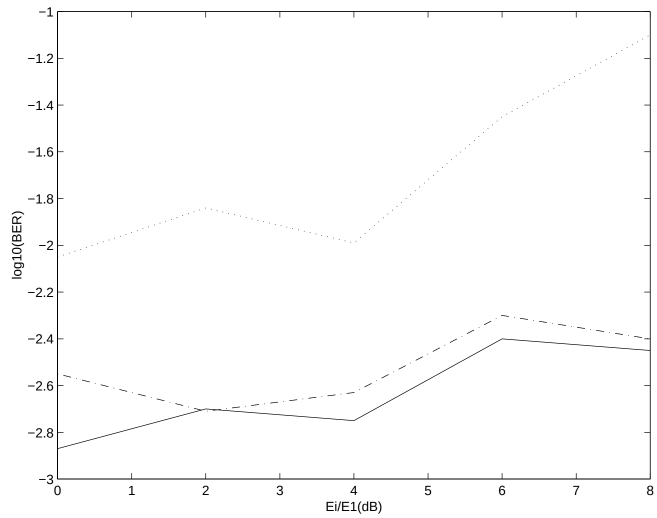

Figure 5.14: Bit error rate versus E_i/E_1 for 5-user channel with the SNR of user 1 fixed at 8dB, where the solid line is OMD, the dash-dot line is the SRN and the dotted line is the MLP.

In the maze problem, a much more complex problem, we showed that we can achieve good results only by training an SRN with a combination of BTT and adaptive learning rates. In addition, we needed to use a special design — a cellular structure — to solve this problem. On the other hand, neither an MLP nor an SRN trained by truncation could solve this problem.

In the multiuser detection problem, we have investigated the comparison of the abilities of the SRN and MLP to provide fast suboptimal solutions to hard combinatorial optimization problems in CDMA systems. Both the SRN receiver and the MLP have been evaluated via extensive simulations. It was shown that the SRN receiver exceeds the performance of the MLP receiver, and approaches the performance of the OMD.

A paper based on part of the work reported in this thesis has been accepted by the *Journal on Mathematical Modeling and Scientific Computing.*

Now that it has been proven that neural networks can solve these kinds of problems, the next step in research is to consider several variations of these problems in order to demonstrate generalization ability and the ability to solve optimization problems where the J function is not known, or not exactly computable.

Appendix A

Appendix: The program of the maze problem using SRN trained by BTT

```
/****************************************************************/
/* The program of the maze problem using SRN trained by BTT;      */
/* Using the SRN trained by BTT to learn the optimal path of a 5*5*/
/*maze;   */
/* Learning Rate Adaptive  --- Lr_Ws,Lr_W,Lr_ww;   */
/* When change line 140 and 141 into p=0 then the program will be */
/*MLP;   */
/* When change line 237 into F_x[N+i]=0 then the program will be  */
/*the SRN trained by Truncation.*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
```

```
void F_NET2(double F_Yhat, double W[30][30],double x[30],int n,
int m,int N, double F_W[30][30], double F_net[30],double F_Ws[30],
double Ws,double F_x[30]);

void NET(double W[30][30],double x[30],double ww[30],
                int n, int m, int N, double Yhat[30]);
void synthesis(int B[30][30],int A[30][30],int n1,int n2);
void pweight(double Ws,double F_Ws_T,double ww[30],
double F_net_T[30],double W[30][30], double F_W_T[30][30],
int n,int N,int m);

int minimum(int s,int t,int u,int v);
int min(int k,int l);
double f(double x);

void main()
{
        int i,j,it,iz,ix,iy,lt,m,maxt,n,n1,n2,nn,nm,N,p,q,po,t,TT;
        int A[30][30],B[30][30];
        double a,b,dot,e,e1,e2,es,mu,s,sum,F_Ws_T,Ws,F_Yhat,wi;
        double W[30][30],x[30],ww_O[50],Ws_O,W_O[50][50],F_Ws_O;
        double F_net_T[30],F_Ws[30],F_W_O[30][30],F_W[30][30];
        double F_W_T[30][30],F_net[30],ww[30],yy[21][12][8][8];
        double Yhat[30],F_y[21][12][8][8],F_x[30],F_Jhat[30][30];
```

```
double S_F_W1,S_F_W2,Lr_W,S_F_net1,S_F_net2,Lr_ww,Lr_Ws;
double y[50][50],F_net_O[50], F_Ws1, F_Ws2;
FILE *f;

/* Number of inputs,neurons and output:7,3,1; */
/* 'n' is the number of the active neurons; */
/* 'm' and 'N' both are the number of inputs; */
/* 'nm' is the number of memory is: 5; */
/* 'nn+1'*'nn+1' is the size of the maze'; */
/* 'TT' is the number of trials; */
/* 'lt' is the number of the interval time; */
/* 'maxt' is the max number for T in figure[8]; */
/* Lr-Ws,Lr_ww and Lr_W are the learning rates for */
/* Ws,ww and W respectively. */

a=0.9; b=0.2;
n=5;m=11;N=11;nn=6;nm=5;TT=30000;lt=50;maxt=20;wi=25;Ws=40;
e=0;po=pow(2,31) -1;

/* Initial values of Old  */

F_Ws_O=1;
for (i=m+1;i<N+n+1;i++){
    for (j=1;j<i;j++)
        F_W_O[i][j]=1;
```

```
    F_net_O[i]=1;
}
Lr_W=Lr_ww=Lr_Ws=10;

/* Initial values of weights */

for (i=1;i<N+n+1;i++)
    x[i]=0;
for (i=m+1;i<N+n+1;i++)
    for (j=0;j<i;j++){
        srand(rand());
        W[i][j]=0.2091;
    }
for (i=m+1;i<N+n+1;i++){
     srand(rand());
     ww[i]=0.00678;
}

/* Input Maze */

n2=5*5;
n1=5*5-1;
for (i=0;i<7;i++)
    for(j=0;j<7;j++){
  if ((i==0)||(j==0)||(i==6)||(j==6))
```

```
          B[i][j]=n2;
        else
                B[i][j]=n1;
    }
/* Generate Obstacle */

B[2][2]=B[3][3]=B[4][4]=n2;

/* Generate Start */

B[2][4]=1;

for (i=0;i<7;i++)
        for (j=0;j<7;j++){
            A[i][j]=0;
        }

/* Desired outputs */

synthesis(B,A,n1,n2);

if ((f=fopen("results5","w"))==NULL) {
   printf("Cannot open file");
   exit(1);
}
```

```
/* Learning Pattern */

for (t=0;t<TT;t++){
    for (i=m+1;i<N+n+1;i++){
        for (j=1;j<i;j++){
               F_W_T[i][j]=0;
        }
        F_net_T[i]=0;
    }
    for (i=1;i<n;i++)
        for (ix=0;ix<nn+1;ix++)
            for (iy=0;iy<nn+1;iy++)
                yy[0][i][ix][iy]=-1;
        for (ix=0;ix<nn+1;ix++)
            for (iy=0;iy<nn+1;iy++)
                yy[0][n][ix][iy]=0;

    e=F_Ws_T=s=0;

   /*********************************************/
   /* If the next two lines are changed into p=0 */
   /* then it is MLP                             */

   p=(t/lt)+1;
```

```
p=(p<maxt ? p:maxt);
for (q=0;q<p+1;q++){
    e=0;
    for (ix=0;ix<nn+1;ix++)
        for (iy=0;iy<nn+1;iy++){
            if (B[ix][iy]==25)
               x[1]=B[ix][iy];
            else if (B[ix][iy]!=1)
                    x[1]=0;
            x[2]=1;
            if (ix!=0)
               x[3]=yy[q][1][ix-1][iy];
            else
               x[3]=yy[q][1][nn][iy];
            if (iy!=0)
               x[4]=yy[q][1][ix][iy-1];
            else
               x[4]=yy[q][1][ix][nn];
            if (ix!=nn)
               x[5]=yy[q][1][ix+1][iy];
            else
               x[5]=yy[q][1][0][iy];
            if (iy!=nn)
               x[6]=yy[q][1][ix][iy+1];
            else
```

```
              x[6]=yy[q][1][ix][0];
           for (i=1;i<n+1;i++)
               x[6+i]=yy[q][i][ix][iy];
           NET(W,x,ww,n,m,N,Yhat);
           for (i=1;i<n+1;i++)
               yy[q+1][i][ix][iy]=Yhat[i];
    }
}
e=0;
for (ix=0;ix<nn+1;ix++)
    for (iy=0;iy<nn+1;iy++){
        if (t==(TT-1))
           y[ix][iy]=yy[p+1][n][ix][iy];
        if (B[ix][iy]!=25)
           F_Jhat[ix][iy]=Ws*yy[p+1][n][ix][iy]
                           -A[ix][iy];
        else
           F_Jhat[ix][iy]=0;
           e+=F_Jhat[ix][iy]*F_Jhat[ix][iy];
    }
    printf("\n t e %d %e",t,e);
    fprintf(f,"\n%d %e",t,e);

/* Initialize F_y */
for (q=1;q<21;q++)
```

```
    for (ix=0;ix<nn+1;ix++)
        for (iy=0;iy<nn+1;iy++)
            for (i=1;i<n+1;i++)
                F_y[q][i][ix][iy]=0;
for (q=p;q>-1;q--){
    for (ix=0;ix<nn+1;ix++)
        for (iy=0;iy<nn+1;iy++){
            if (B[ix][iy]==25)
               x[1]=B[ix][iy];
            else if (B[ix][iy]!=1)
                    x[1]=0;
            x[2]=1;
            if (ix!=0)
               x[3]=yy[q][1][ix-1][iy];
            else
               x[3]=yy[q][1][nn][iy];
            if (iy!=0)
               x[4]=yy[q][1][ix][iy-1];
            else
               x[4]=yy[q][1][ix][nn];
            if (ix!=nn)
               x[5]=yy[q][1][ix+1][iy];
            else
               x[5]=yy[q][1][0][iy];
            if (iy!=nn)
```

```
   x[6]=yy[q][1][ix][iy+1];
else
   x[6]=yy[q][1][ix][0];
for (i=1;i<n+1;i++)
    x[6+i]=yy[q][i][ix][iy];
NET(W,x,ww,n,m,N,Yhat);
if (q==p){
   F_Yhat=F_Jhat[ix][iy];
   for (i=1;i<n+1;i++)
       F_x[N+i]=0;
}
else {
   F_Yhat=0;
   for (i=1;i<n+1;i++)
       F_x[N+i]=F_y[q+1][i][ix][iy];
       /*if  F_x[N+i] = 0; truncation */
}
F_NET2(F_Yhat,W,x,n,m,N,F_W,F_net,
                     F_Ws,Ws,F_x);
if (ix!=0)
   F_y[q][1][ix-1][iy]+=F_x[3];
else
   F_y[q][1][nn][iy]+=F_x[3];
if (iy!=0)
   F_y[q][1][ix][iy-1]+=F_x[4];
```

```
            else
               F_y[q][1][ix][nn]+=F_x[4];
            if (ix!=nn)
               F_y[q][1][ix+1][iy]+=F_x[5];
            else
               F_y[q][1][0][iy]+=F_x[5];
            if (iy!=nn)
               F_y[q][1][ix][iy+1]+=F_x[6];
            else
               F_y[q][1][ix][0]+=F_x[6];
            for (i=1;i<n+1;i++)
                F_y[q][i][ix][iy]+=F_x[6+i];


            if (q==p) F_Ws_T+=F_Ws[1];


            for (i=m+1;i<N+n+1;i++){
                for (j=1;j<i;j++){
                    F_W_T[i][j]+=F_W[i][j];
                }
                F_net_T[i]+=F_net[i];
            }
        }
}
dot=0;
for (i=m+1;i<N+n+1;i++)
```

```
    for (j=1;j<i;j++){
        dot+=F_W_O[i][j]*F_W_T[i][j];
    }



S_F_W1=S_F_W2=0;
for (i=m+1;i<N+n+1;i++)
    for (j=1;j<i;j++){
        S_F_W1 += F_W_O[i][j] * F_W_T[i][j];
        S_F_W2 += F_W_O[i][j] * F_W_O[i][j];
        s+=F_W_T[i][j]*F_W_T[i][j];
    }
if ((S_F_W1>S_F_W2) || (S_F_W1==S_F_W2))
   Lr_W=Lr_W*(a+b);
else if (S_F_W1<(-2)*S_F_W2)
        Lr_W=Lr_W*(a-2*b);
     else
        Lr_W=Lr_W*(a+b*(S_F_W1/S_F_W2));

S_F_net1=S_F_net2=0;
for (i=m+1;i<N+n+1;i++){
    s+=F_net_T[i]*F_net_T[i];
    S_F_net1 +=F_net_O[i] *F_net_T[i];
    S_F_net2 +=F_net_O[i] *F_net_O[i];
}
```

```
if ((S_F_net1>S_F_net2) || (S_F_net1==S_F_net2))
   Lr_ww=Lr_ww*(a+b);
else if (S_F_net1<(-2)*S_F_net2)
        Lr_ww=Lr_ww*(a-2*b);
     else
        Lr_ww=Lr_ww*(a+b*(S_F_net1/S_F_net2));
F_Ws1=F_Ws_O*F_Ws_T;
F_Ws2=F_Ws_O*F_Ws_O;
if ((F_Ws1>F_Ws2) || (F_Ws1==F_Ws2))
   Lr_Ws=Lr_Ws*(a+b);
else if (F_Ws1<(-2)*F_Ws2)
        Lr_Ws=Lr_Ws*(a-2*b);
     else
        Lr_Ws=Lr_Ws*(a+b*(F_Ws1/F_Ws2));
s+=F_Ws_T*F_Ws_T;
es=e/s;
if ((e-Lr_W*s)<0)
   Lr_W=Lr_W*es;
if ((e-Lr_ww*s)<0)
   Lr_ww=Lr_ww*es;
for (i=m+1;i<N+n+1;i++){
    for (j=1;j<i;j++){
        W_O[i][j]=W[i][j];
        W[i][j]-=Lr_W*F_W_T[i][j];
    }
```

```
            ww_O[i]=ww[i];
            ww[i]-=Lr_ww*F_net_T[i];
        }
        if ((e-Lr_Ws*s)<0)
           Lr_Ws=Lr_Ws*es;
        Ws_O=Ws;
        Ws-=Lr_Ws*F_Ws_T;

        sum=0;
        for (i=m+1;i<N+n+1;i++){
            for (j=1;j<i;j++){
                F_W_O[i][j]=F_W_T[i][j];
            }
            F_net_O[i]=F_net_T[i];
        }
        F_Ws_O=F_Ws_T;
   }
   fclose(f);
}


void synthesis(int B[30][30],int A[30][30],int n1,int n2)
{
        int k,mini,no,i,j;
```

```
        /*  Initialization */

        k=0;
        for (i=0;i<7;i++)
            for (j=0;j<7;j++){
                 A[i][j]=B[i][j];
            }

/* Searching the optimal path */

        /* Calculating the Utility */
        no = n2-3-1;
        while (k!=no){
           k=0;
           for (i=1;i<6;i++)
               for (j=1;j<6;j++){
                  mini = 1 +
           minimum(A[i-1][j],A[i][j-1],A[i+1][j],A[i][j+1]);
                  if ((A[i][j]!=n2) && (A[i][j]!=1)){
                     if ((A[i][j]==mini) && (A[i][j]!=n1))
                        k++;
                  else
                     if (mini!=n2)
                        A[i][j]=mini;
               }
```

```
            }
        }
}

/* minimum: return the minimum value */

int minimum(int s,int t,int u,int v)

{
        int mini;
        mini=0;
        mini=min(min(min(s,t),u),v);
        return mini;
}

void NET(double W[30][30],double x[30],double ww[30],
                int n, int m, int N, double Yhat[30])

{
        int i,j;
        double net;
        for (i=m+1;i<N+n+1;i++)
        {
                net=0;
                for (j=1;j<i;j++){
```

```
                net += W[i][j] * x[j];
            }
            x[i]=f(net+ww[i]);
        }
        for (i=1;i<n+1;i++){
            Yhat[i]=x[i+N];
        }
}


double f(double x)
{
        double z;
        z=(1-exp(-x))/(1+exp(-x));
        return z;
}

void F_NET2(double F_Yhat,double W[30][30],double x[30],int n,
int m,int N,double F_W[30][30],double F_net[30],double F_Ws[30],
double Ws, double F_x[30])
/* This subroutine calculates the F_W terms needed to adapt a
fully connected; */
/*generalized MLP. It does not backpropagate through the network
and it does not permit the switching off of weights. */
{
```

```
        int i,j;
        for (i=1;i<N+1;i++)
                F_x[i]=0;
        F_x[N+n]+=F_Yhat*Ws;
        F_Ws[1]=F_Yhat*x[N+n];
        F_net[N+n]=F_x[N+n]*(1-x[N+n]*x[N+n])*0.5;
        for (j=1;j<N+n;j++)
                F_W[N+n][j]=F_net[N+n]*x[j];
        for (i=N+n-1;i>m;i--)
        {
                for (j=i+1;j<N+n+1;j++)
                {
                        F_x[i]+=W[j][i]*F_net[j];
                }
                F_net[i]=F_x[i]*(1-x[i]*x[i])*0.5;
                for (j=1;j<i;j++)
                {
                        F_W[i][j]=F_net[i]*x[j];
                }
        }
        for (i=m;i>0;i--)
            for (j=m+1;j<N+n+1;j++)
                F_x[i]+=W[j][i]*F_net[j];
}
```

```
int min(int k, int l)
{
        int r;
        if (k>l) r=l;
        else r=k;
        return r;
}

void pweight(double Ws,double F_Ws_T,double ww[30],
double F_net_T[30],double W[30][30], double F_W_T[30][30],
int n,int N,int m)
{
        int i,j;
        for (i=m+1;i<N+n+1;i++){
            for (j=1;j<i;j++){
              printf("\n W[i][j] F_W_T %e %e",W[i][j],F_W_T[i][j]);
            }
            printf("\n ww F_net_T %e %e",ww[i],F_net_T[i]);
        }
        printf("\n Ws F_Ws_T %e %e",Ws,F_Ws_T);

}
```

BIBLIOGRAPHY

[1] E. D. Sontag, *Feedback stabilization using two-hidden-layer nets*, IEEE Trans. Neural Networks, Vol. 3, No.6, 1992.

[2] P. Werbos, *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*, Wiley, 1994.

[3] I. Guyon, I. Poujaud, L. Personnaz, G. Dreyfus, J. Denker, and Y. Le Cun, *Comparing different neural network architectures for classifying handwritten digits*, Proceedings of the IEEE International Joint Conference on Neural Networks, June 1989.

[4] Von der Malsburg, C.& Schneider, W. Biol. *Cybernetic*, Vol. 54, pp. 29-40, 1986.

[5] W. Miller, R. Sutton & P. Werbos (eds.), *Neural Networks for Control*, MIT Press, 1990.

[6] V. B. Brooks, *The Neural Basis of Motor Control*, Oxford press.

[7] K. Pribram, *Brain and Perception: Holonomy and Structure in Figural Processing*, Erlbaum, 1991.

[8] H. Chang, W.J. Freeman, *Parameter optimization in models of the olfactory neural system*, Neural Networks, Vol. 9, No. 1, pp 1-14,1996.

[9] D.White & D.Sofge (eds.), *Handbook of Intelligent Control: Neural, Adaptive and Fuzzy Approaches*, Van Nostrand, 1992.

[10] P. Werbos, *The brain as a neurocontroller: New hypotheses and new experimental possibilities*, In K.Pribram (eds.), Origins: Brain and Self-Organization, Erlbaum, 1994.

[11] P. Werbos, *Learning in the brain: engineering interpretation*, In K.Pribram, (eds.), Learning as Self-organization, Erlbaum, 1996.

[12] D. Prokhorov, R. Santiago & D. Wunsch, *Adaptive critic designs: a case study for neurocontrol*, Neural Networks, Vol.8, No.9, 1995.

[13] D.O.Hebb, *Organization of Behavior*, Wiley, New York, 1949.

[14] J. Hopfield and D. Tank, *Computing with neural circuits: A model*, Science, Vol. 233, pp. 625-633, 1986.

[15] S. Grossberg, *The Adaptive brain I*, North-Holland, 1987.

[16] P. Werbos, *Optimization methods for brain-like intelligent control*, IEEE Conference on Decision and Control, 1995.

[17] P. Werbos, *Beyond regression: new tools for predictions and analysis in the behavioral science*, Ph.D. dissertation, Committee on Applied Mathematics, Harvard University, Cambridge, MA, Nov. 1974.

[18] D. Rumelhart, G. Hinton and R. Williams, *Learning internal representations by error propagation*, in D.Rumelhart and J.McClelland (eds.), Parallel Distributed Processing, Vol.1, MIT Press, 1986.

[19] L. Fausett, *Fundamentals of Neural Networks: architectures, algorithms and applications*, Prentice Hall, 1994.

[20] P. Werbos, *Generalization of backpropagation with application to a recurrent gas market model, neural networks*, Vol. 1, pp. 339-365, 1988.

[21] L. B. Almeida, *A learning rule for asynchronous perceptrons with feedback in a combinatorial environment*, Proceedings of the IEEE International Conference on Neural Networks, 1987.

[22] F. J. Pineda, *Generalization of backpropagation to recurrent and higher oder networks*, Proceedings of the IEEE International Conference on Neural Information Processing Systems, 1987.

[23] P. Werbos, *Supervised learning: can it escape its local minimum*, WCNN93 Proceedings, Erlbaum, 1993. Reprinted in V. Roychowdhury et al (eds.), Theoretical Advances in Neural Computation and Learning, Kluwer, 1994.

[24] P. Houillon and A. Caron, *Planar robot control in cluttered space by artificial neural network*, Math Modeling and Science Computing, Vol. 2, pp. 498–502, 1993.

[25] M. L. Minsky and S. A. Papert, *Perceptrons*, MIT Press, 1990, expanded edition.

[26] T. Maxwell, L. Giles and Y. C. Lee, *Generalization in neural networks: the contiguity problem*, IEEE First International Conference on Neural Networks, 1987.

[27] P.K.H. Phua AND S.B.W. Chew, *Symmetric rank-one update and quasi-Newton methods*, Optimization Techniques and Applications, Proceedings of the International Conference on Optimization Techniques and Applications, K.H. Phua et al., eds., World Scientific, 1992, Singapore, pp. 52–63.

[28] R. Howard, *Dynamic Programming and Markov Processes*, MIT press, Cambridge, MA, 1960.

[29] P. Werbos, *Neural networks for control and system identification*, IEEE Conference on Decision and Control (Florida), IEEE, New York, 1989.

[30] A. Barron, *Asymptotically optimal functional estimation by minimum complexity criteria*, Proceedings of 1994 IEEE International Symposium on Information Theory, IEEE, New York, 1994.

[31] J. S. Baras and N. S. Patel, *Information state for robust control of set-valued discrete time systems*, Proceedings of 34th Conference on Decision and Control, IEEE, 1995, p. 2302.

[32] Poor, H.V. and Sergio Verdú, *Probability of Error in MMSE Multiuser Detection*, IEEE Trans. Infom. Theory, Vol. 43, pp. 858-871, May 1997.

[33] Kechriotis, G. I. and Manolakos S., *Hopfield Neural Network Implementation of the Optimal CDMA Multiuser Detector*, IEEE Trans. Neural Networks, Vol. 7, pp.131-141, Jan. 1996.

[34] Miyajima, T. etc., *On the Multiuser Detection Using a Neural Network in Code-Division Multiple-Access Communications*, IEICE Trans. Commun., Vol. E76-B, Aug. 1993.

[35] Aazhang, B., Paris B.-P., and Orsak G.C., *Neural Networks for Multiuser Detection in Code-Division Multiple-Access Communications*, IEEE Trans. Commun., Vol 40, pp 1212 -1222, July 1992.

[36] He,G., Tang P. and Pang, X., *Neural Network Approaches to Implementation of Optimum Multiuser Detectors in Code-Division Multiple-Access Channels*, Intl. Journ. of Electronics, Vol. 80, pp.425-431, 1996.

[37] Verdú, S., *Minimum Probability of Error for Asynchronous Gaussian Multiple-Access Channels*, IEEE Trans. Infom. Theory, Vol. IT-32, Jan. 1986.

[38] Duel-Hallen, A., etc, Multiuser Detection for CDMA Systems, IEEE Personal Communications, April 1995.

[39] Lippmann, R.P., *An Introduction to Computing with Neural Nets*, IEEE ASSP Magazine, Vol. 4, April 187.

[40] Hopfield, J.J., *Neurons with graded response have collective computational properties like those of two-state neurons*, Proc. Nat. Acad. Sci. USA, Vol. 81, pp. 3088-3092, 1984.

[41] Bertsekas, D.P., *Dynamic programming: deterministic and stochastic models*, Prentice-Hall, 1987.

[42] Laveen, K., *On pattern, categories, and alternate realities*, University of Maryland, 1993.