

# **CONNECTIONIST LEARNING PROCEDURES**

Dissertation submitted to  
Jawaharlal Nehru University  
in partial fulfilment of the requirements  
for the award of the Degree of  
**MASTER OF TECHNOLOGY**  
in  
**COMPUTER SCIENCE AND TECHNOLOGY**

by  
**JYOTI PRAKASH HANDIQUE**

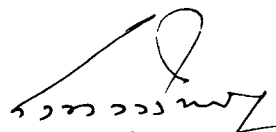
**SCHOOL OF COMPUTER AND SYSTEMS SCIENCES  
JAWAHARLAL NEHRU UNIVERSITY  
NEW DELHI-110 067,  
JANUARY 1992**


## CERTIFICATE

This is to certify that the thesis entitled "**CONNECTIONIST LEARNING PROCEDURES**", being submitted by me to Jawaharlal Nehru University in partial fulfilment of the requirements for the award of the degree of Master of Technology, is a record of original work done by me under the supervision of Dr. K. K. Bhardwaj, Prof. School of Computer and Systems Sciences, during the Monsoon semester, 1991.

The results reported in this thesis have not been submitted in part or full to any other University or Institution for the award of any degree etc.

  
(JYOTI PRAKASH HANDIQUE)

  
Dr. R. G. Gupta 31/12/91  
Professor & Dean,  
SC&SS, J.N.U.,  
New Delhi - 110 067.

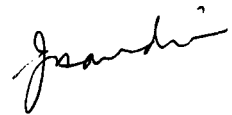
  
31-12-91  
Dr. K. K. Bhardwaj  
Professor,  
SC&SS, J.N.U.,  
New Delhi - 110 067.

## ACKNOWLEDGEMENTS

I wish to express my sincere and heartfelt gratitude to Dr. K. K. Bharadwaj, Professor, School of Computer and Systems Sciences, Jawaharlal Nehru University, for the unfailing support he has provided through out. In all respects, I am very grateful for the patience he has exhibited and for the time he has spent with me discussing the problem. It would have been impossible for me to come out successfully without his constant guidance.

I extend my thanks to Prof. R. G. Gupta, Dean, School of Computer and System Sciences, JNU for providing me the opportunity to undertake this project. I would also like to thank the authorities of our school for providing me the necessary facilities to complete my project.

I am grateful to all the friends, whom I had approached for their handwriting samples, for their prompt response. I acknowledge and thank each and everyone of those who, directly or indirectly, helped me in this work.

A handwritten signature in black ink, appearing to read 'J. K. K.' or similar, located at the bottom right of the page.

## ABSTRACT

In this work, a study of various neural network learning procedures has been done. In particular, the Recursive Least Squares (RLS) algorithm as applied to a two layer linear combiner network is studied in detail. As an application of neural networks in adaptive pattern recognition the RLS algorithm for linear combiner networks is made use of in developing an algorithm for handwritten character recognition. A prototype implementation of the algorithm developed is also given using PASCAL and experimental results discussed.

## CONTENTS

Chapter 1	INTRODUCTION	1
	Neural or connectionist networks	2
	Different models of neural networks	4
	Properties	6
	Learning in connectionist models	8
	Applications	9
Chapter 2	CONNECTIONIST LEARNING PROCEDURES	11
	Linear associator	12
	Nonlinear associative nets	12
	Simple supervised learning procedures	13
	Perceptron learning algorithm	14
	Back propagation in multilayer networks	15
	Boltzmann machine	16
	Competitive learning	17
	Reinforcement learning procedures	17
	Speed of learning	18
Chapter 3	THE RECURSIVE LEAST SQUARES ALGORITHM	20
	Adaptive combiner	20
	Analytical derivation of	
	Least squares algorithm	21
	Recursive least squares	26
	Computational cost of the RLS algorithm	29
Chapter 4	NEURAL NETWORKS AND PATTERN RECOGNITION	30
	Decision surface	31
	Similarity, Distance and compactness	32
	Neural networks and	
	Adaptive pattern recognition	34
	Linear separability	34
	Non linear separability	35
	Layered neural nets	36
	Handwritten letter classification	36

Chapter 5	IMPLEMENTATION OF THE RLS ALGORITHM FOR HANDWRITTEN CHARACTER RECOGNITION	38
	The network	39
	Supervised learning	39
	Implementation of the algorithm	40
	Results	42
	CONCLUSIONS	43
	BIBLIOGRAPHY	44

## **CHAPTER 1**

### **INTRODUCTION**

---

## CHAPTER 1

### INTRODUCTION

Conventional digital computers are extremely good at numerical computation and executing sequences of instructions, that has been precisely formulated for them. On the other hand, the human brain performs well at such tasks as vision, speech, information retrieval, and complex pattern recognition in the presence of noisy and distorted data, and common sense reasoning. Somehow, the structure of the human brain is better suited for such kind of tasks and not suited for tasks such as numerical computations. The human brain is a naturally occurring example of an intelligent machine. It follows that, one natural idea for Artificial Intelligence is to simulate the functioning of the human brain on a computer.

In the 1950s work in AI started with two goals: design intelligent machines or programs and understand human intelligence. The possible inter-twining of these two goals was particularly stressed by Von Neumann, who introduced the theory of automata to study the logical differences and the similarities between Natural and Artificial machines. McCulloch and Pitts, in their attempts to simulate the nervous cells by artificial automata -the formal neurons - had shown in 1943 that a network of such formal neurons was capable of simulating a Turing Machine. Later on, Rosenblatt, Minsky and Pappert, in their work on perceptrons, tried to push forward the model of formal neurons so as to make them learn how to solve problems. However, research in



such kinds of networks came to a virtual halt in 1970s, when they were found to be very weak computationally.

The popularity of such network models in AI has taken wide swings, ranging from extreme enthusiasm in the 1960s to utter anathema in the 70s, when they were found to be weak. But currently, there has been an explosion of interest in these approaches. Many reasons can be pointed out for this, including the emergence of fast digital computers, interest in building massively parallel computers and most importantly, the discovery of powerful learning algorithms. Moreover, it has shown promises as a research tool in many disciplines, and engineers, mathematicians, worldwide are turning towards such models as a possible alternative to their conventional research techniques.

#### **Neural or connectionist networks**

Neural networks, also called Connectionist networks, are based on neurological models. A biological neuron basically consists of a cell body, dendrites axons, as shown in fig 1.1(a). The cell body, which is called the 'soma', performs complicated chemical processes, such as summation and firing with respect to a threshold level. The inputs for a cell body are transmitted through the dendrites, while the output signals are carried to other cells through axons. The electrical signal of an axon connects to a dendrite through a special contact, called a synapse. In general, the neuron performs a simple threshold function. When the potential inside the cell body is larger than the threshold value, the neuron fires. The normal firing rate is quite low, which is typically a few hundred occurrences per second.

It is estimated that the human brain has approximately  $10^{11}$  neurons and  $10^{14}$  synapses. In the artificial neural systems, the neuron and the synapses are configured as the processing elements and the connection strength respectively. Various features of the artificial neural systems are determined by the function of the neuron and the interconnection pattern.

The artificial neural network can be characterized by the following properties:

- (1) network architecture,
- (2) retrieving process,
- (3) learning rule and
- (4) training data.

The network architecture provides the most distinguished feature. The grouped neurons, which are arranged into a disjointed structure are called layer. Fig 1.2 shows several architectures which include two-layer/ multi layer and feedforward / feedback networks. The neuron transfer function and the threshold voltage characterize the retrieving process of an artificial neural network. Specific mathematical functions including sigmoid, step, Gaussian, Boltzmann functions are widely used to model the neuron transfer function. The nonlinear transfer function decides the information propagation properties at the neural retrieving process. The retrieving process can operate in either synchronous or asynchronous mode. In the synchronous mode, all the neuron outputs are updated simultaneously. Conversely, the neuron updating process in the asynchronous mode is random and independent of the other neurons. Most artificial neural networks in software computation operate synchronously, while the

biological neural networks operate in the fully asynchronous mode. The training procedures are divided into two categories: supervised and unsupervised learning. In supervised learning, the synapse weightings are tuned by the difference between the retrieving patterns and the expected patterns. In unsupervised learning, the network classifies the inputs without references. The neural networks using unsupervised learning can detect the pattern regularities. The widely used learning rules include Hebb rule, Delta rule, competitive learning rule and their derivatives, which will be described in Chapter 2. In general, the input signal for an artificial neural network can be discrete or continuous values.

#### **Different models of neural networks**

After McCulloch and Pitts introduced the abstract neuron model for performing a simple task, in 1943, the neural network study began. F. Rosenblatt developed the perceptron, which sparked a great amount of research interest in neurocomputing. The Perceptron is a two layer network for pattern classification. Initially, the perceptron demonstrated an optical pattern recognition when inputs of the system were connected to a grid of Photocells. The input signals are then transferred to the neural layer with randomly weighted connections. The neural network performed successfully with application of Hebb learning rule. The major limitation, pointed out by Minsky and Pappert, is that the perceptron cannot represent an XOR function, so that the perceptron cannot classify complex categories. Multi layer Perceptrons were developed by Rosenblatt to overcome the limitation of the initial

Perceptron.

In the late 50s, the first neural element Adaptive Linear Element (ADALINE) was developed by B. Widrow. The neurons were realised with vacuum tube amplifiers, while synapse weights were manually adjusted with variable resistors. The ADALINE was improved to become the MADALINE, which consists of ADALINES and a two layer variant. These were adapted to a variety of applications, such as speech recognition, character recognition, weather prediction, adaptive control and echo cancellation in communication equipments.

Another major category in neural networks is associative memory. J. Anderson proposed the 'Brain-state-in-a-box' model with his linear associator and Hebb learning rule. The network consists of a layer with feedback, and one postprocessing output layer. Due to the positive feedback architecture and the learning rule, the output is the best-matched pattern from the stored memory for a given input.

In 1982, the presentation of J. Hopfield's paper to the National Academy of Science ignited the neural network study once again. The Hopfield network is basically a two-layer network with feedback. The condition for the synapse weighting is very restricted, while that for the neuron transfer function is very relaxed. Using the energy of Lyapunov's function, Hopfield proved that the network always moved towards a low energy level. Due to the simple architecture and clearly proved dynamics of the network, many hardware implementations and real world applications have been accomplished. The network has been applied to associative memories and many Engineering optimization problems.

The multi layer neural networks are vitalized by the back propagation learning rule. Before the learning rule was developed, the usefulness of the multilayer neural network had been well known but the decision of the synapse weightings was the main problem. The multilayer neural network can be used for various applications including data encoding/decoding, data compression, signal processing, noise filtering, pattern classification and forecasting.

A Boltzmann machine has the network architecture as the Hopfield network, but differs in the stochastic updates and learning properties. The stochastic updates in retrieving and learning processes is based on the simulated annealing technique using the Boltzmann probability function. By decreasing the temperature of the probability function from a high value, the network always finds the global minimum in the energy surface.

The Bidirectional Associative Memory (BAM) designed for optical computing is a generalized Hopfield model to heteroassociative network. BAM has two fully connected central layers and input output buffer layers. The synapses and the neurons in the central two layers are bidirectional. For a given input, the BAM layers oscillate until a stable state is reached. The final stable output is the closest association stored in BAM.

### **Properties**

A neural network model can be described according to their network, cell and dynamic properties as follows.

**Network properties :** A neural network model consists of network of autonomous processing elements called neurons, that are joined

by connection paths as shown in fig. 1.2. Each such connection has a numerical weight  $w_{i,j}$  that roughly corresponds to the influence of cell  $u_i$  on cell  $u_j$ . Positive weight means reinforcement while a negative weight means an inhibition. These weights determine the behaviour of the network.

The neurons are generally arranged in several layers, with an input layer and an output layer and intermediate or hidden layers between them. The input layers have no entering weights. The response of the output cells are taken as the output of the network. The hidden layers add to the power of the network to compute difficult functions, known as unseparable functions.

**Cell properties :** Each cell, or neuron, compute a single numerical cell output or activation. Typically, every cell uses the same algorithm to compute its activation. The activation of a cell is calculated from the activation of the cells directly connected to it and the weights of these connections. Every cell (except for the input cells) computes its new activation  $u_i$  as a function of the weighted sum of the inputs to the cell from directly connected cell as follows:

$$S_i = \sum_{j=0}^n w_{i,j} u_j \quad (1.1)$$

$$u_i = f( S_i ) \quad (1.2)$$

Here  $w_{i,j}$  is the weight associated to the connection from cell  $j$  to cell  $i$ , if  $j$  is not connected to  $i$  then  $w_{i,j}=0$ .  $u_j$  is the activation of cell  $j$ . Here,  $f$  is a nonlinear function, which may be a step function, a sigmoidal function, the Gaussian function or the Boltzmann function. By convention there is a 0th cell, whose

input is always +1, which is connected to every other cell, except the input cells. The corresponding weights  $w_{i,0}$  are called the biases. The biases are merely a constant term added to the sum of the activations in equation (1.1). They are a means to adjust the threshold values of the neurons.

**Dynamic properties :** A connectionist model must specify when a cell computes a new activation value and when the change to that cell's output actually takes place. In some models cells are visited in a fixed order, each cell reevaluating and changing its activation before the next one is visited. In other models, all cells compute their new activation simultaneously and then makes changes to all outputs simultaneously. Still other models pick a cell at random, compute its new activation, and then change the output immediately before any other cell computes its new activation.

### **Learning in connectionist models**

Learning in a connectionist network is a process of adapting the connection weightings in response to the external stimuli. The learning rules were developed with the network architectures. The first learning rule, called Hebb rule, which shows that the network can learn for a certain function, was presented in 1957. This rule requires that if an input and an output are activated at the same time, the weighting between the input and the output are increased. In competitive learning, each neuron competes with others at a given input and the winner adapts to get more strength. This kind of learning, called unsupervised learning, does not need reference data. On the other hand, desired outputs

can be given in the supervised learning approaches. A famous application of the supervised learning is the back propagation for a multilayer network. Many derivatives of this Delta rule are used for efficient learning results. These rules are applied to adjust the connection weights, using the error between the desired output and the actual network output. The goal of these learning rules is to minimize the output error, or some function of it (usually, the mean square error).

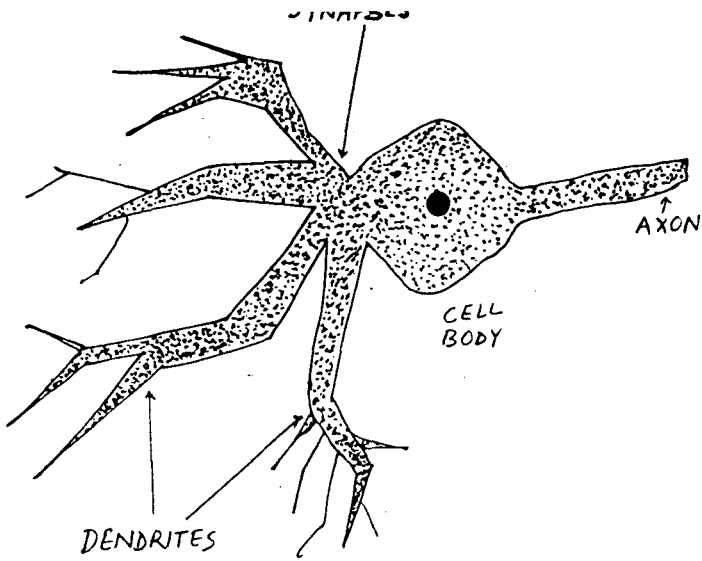
Another type of learning rule that falls between unsupervised learning and supervised learning is reinforcement learning. In this kind of learning, an external observer gives a response as to whether the network response is good or not. The learning rule of Boltzmann machine is based on the stochastic process, which constructs distributed representations of the reference patterns with the simulated annealing technique.

### **Applications**

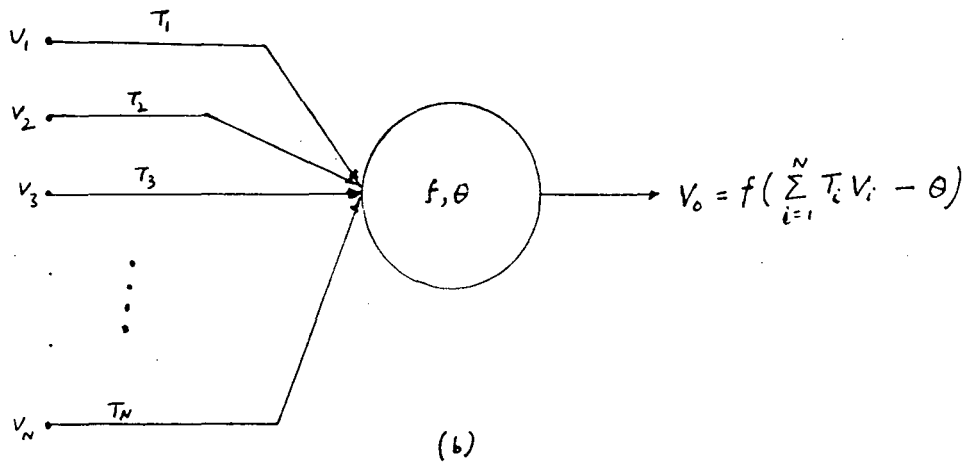
The secret of immense computational power in neural networks is discovered as the parallel processing done by neurons and connections. While each neuron performs simple analog processing at low speed, the rich connectivity of the neurons through synapses provides powerful computational capabilities for the large quantity of data. The data are processed asynchronously in the time domain and spread globally into all network elements. In addition to the parallel processing nature, the network has a self learning capability, which is done by changing the weights of the synapses between the neurons. The self learning capability makes such networks useful in a situation when the training data are sufficient and fault tolerance of a system is necessary. The



immense computational power and the self learning capability give neural networks excellent prospects in image processing, vision understanding, inexact knowledge processing, forecasting, linear/nonlinear programming, scientific optimization and many others. Fig 1.4 shows a block diagram of an advanced neural computing system. Real world signals are converted into discrete form (mainly digital) at the interface block. The neural signal processing system handles the converted signals, and the outputs can be transferred to digital computers for further manipulations. The interface block might function as a data converter and conduct some signal processing. The signals inside a neural signal processor are distributed throughout the whole network. Thus, a small amount of damage in the system does not produce noticeable degradation of the overall system performance. Through self learning procedures, the interconnection weights can be modified, so that the original system performance is retained.



(a)

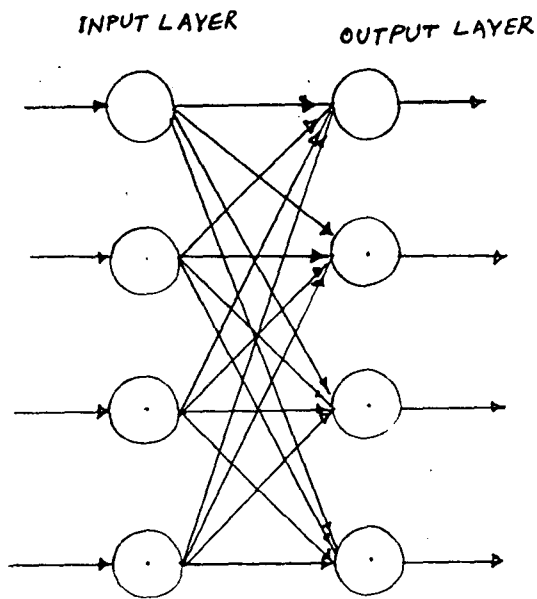


(b)

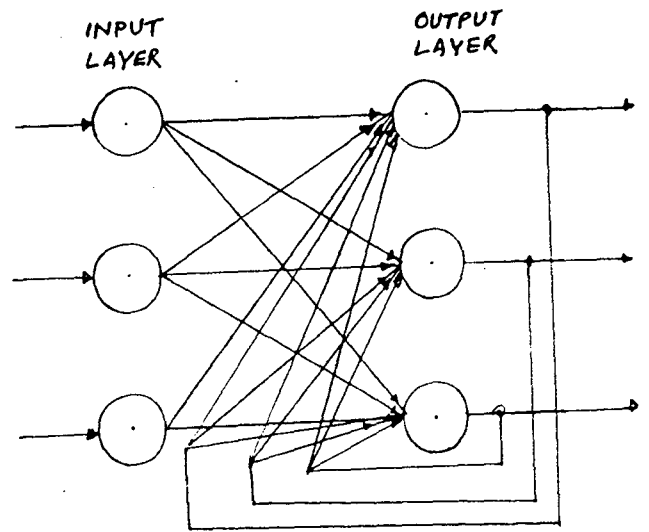
Figure 1.1 Neuron models

(a) Biological neuron model

(b) Artificial neuron model.



(a)



(b)

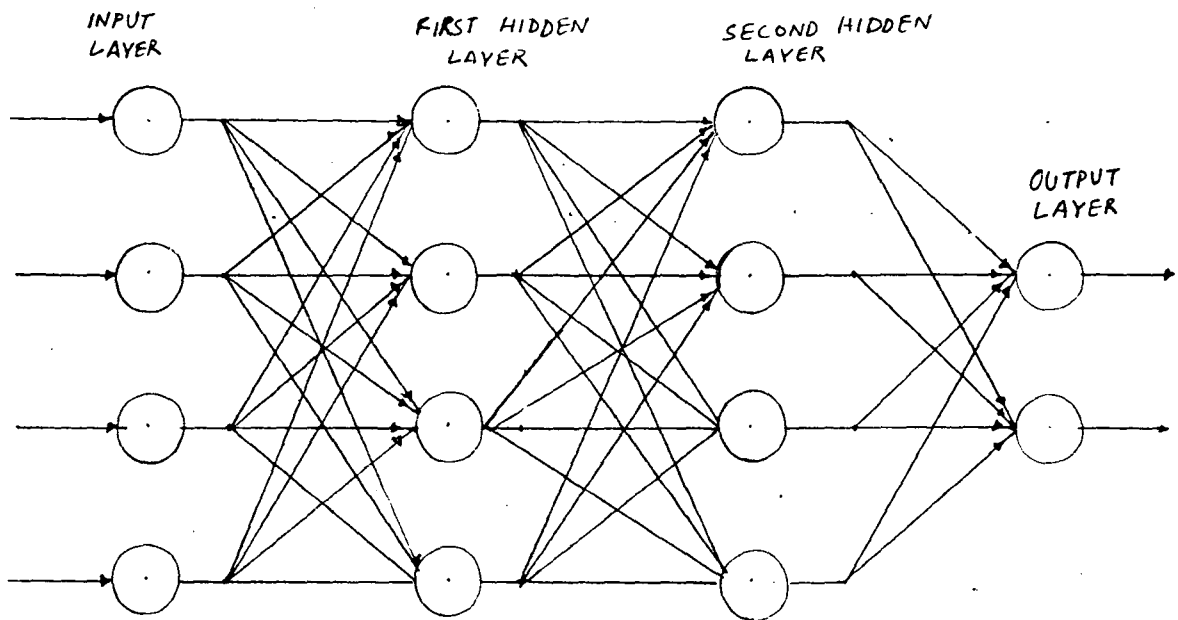
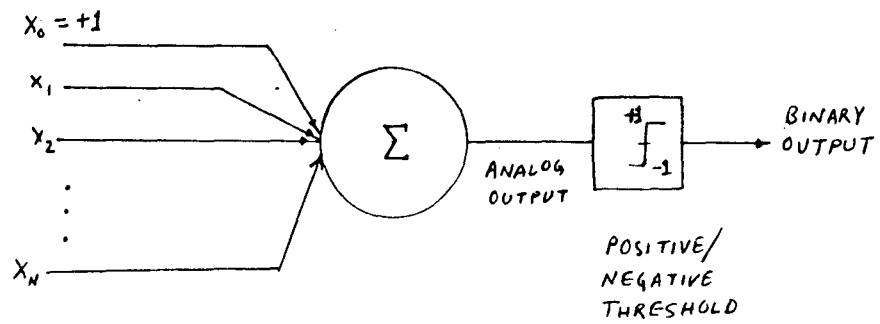


Figure 1.2 Several models of neural networks

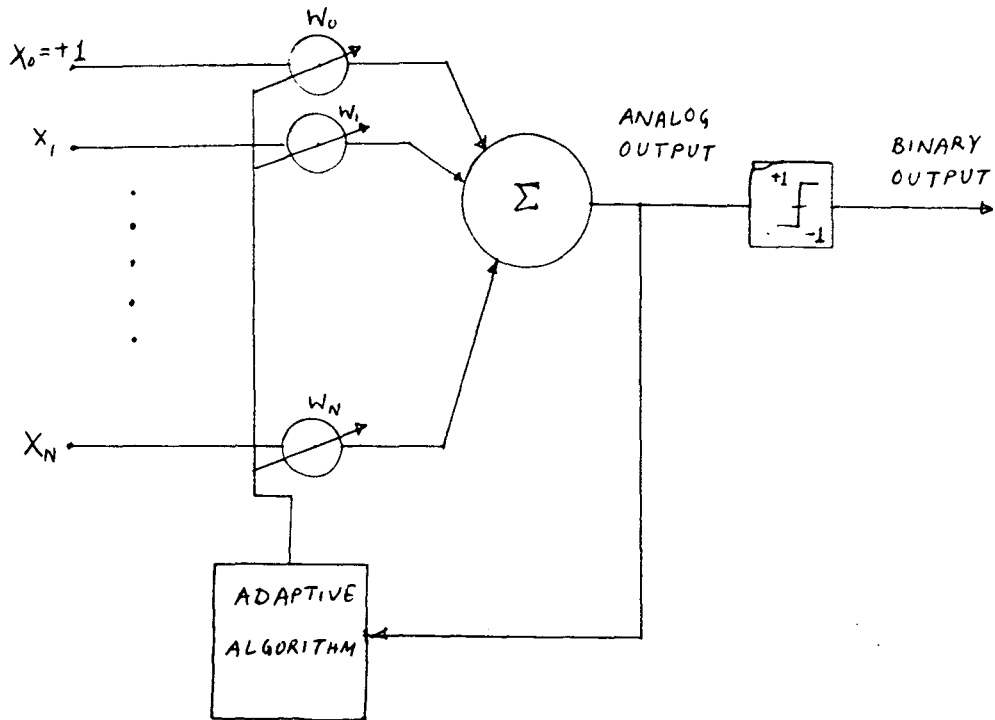
(a) Single layer feed forward

(b) Single layer feedback

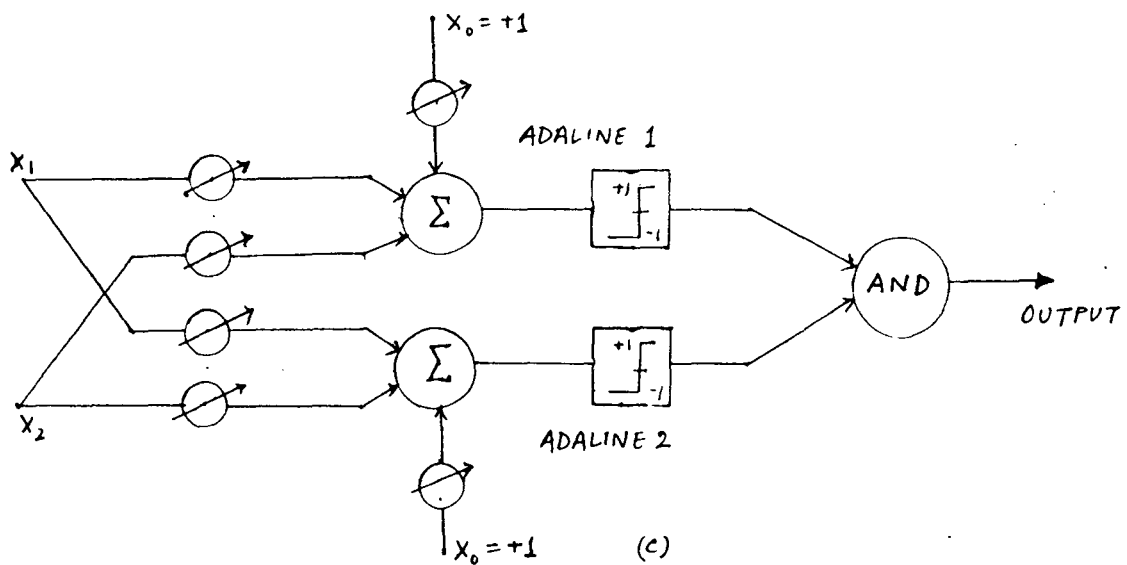
(c) network with hidden layers



(a)



(b)



(c)

Figure 1.3

(a) A Perceptron

(b) ADALINE (Adaptive linear element)

(c) MADALINE

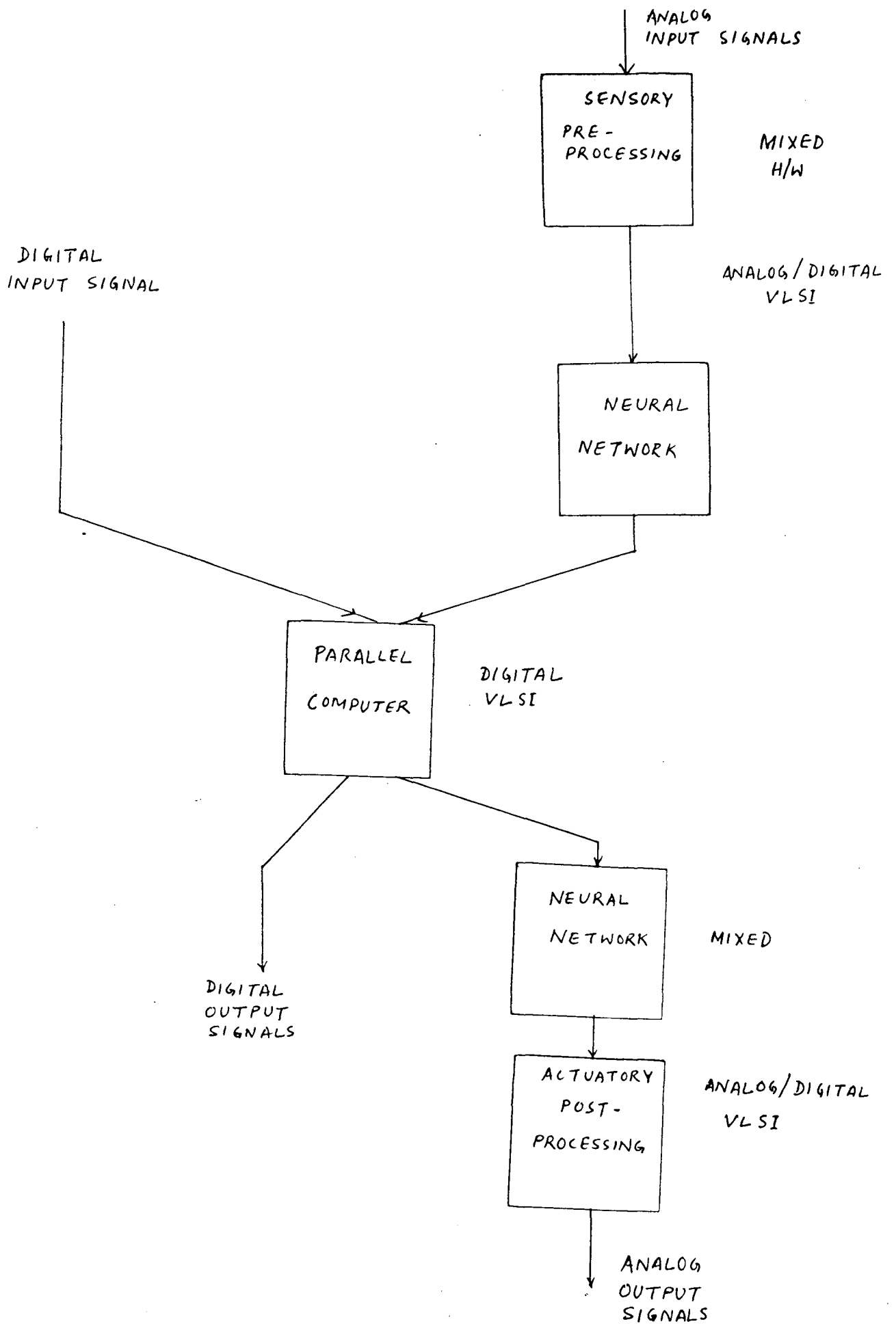


Figure 1.4. Advanced Neural Computing System

## CHAPTER 2

### CONNECTIONIST LEARNING PROCEDURES

---

## CHAPTER 2

### CONNECTIONIST LEARNING PROCEDURES

Learning is the most important part of the neural or the connectionist models. Learning is the process of modifying the values of the weights of the connections and the threshold. Since the weights associated with the connection path determine the behaviour of the network, learning procedures try to adjust the weights in such a way that, the network output approaches the desired values. The learning procedures cannot generate or alter the internal representations, they are limited to forming simple associations between the representations, that are specified externally. Recent researches have led to a variety of powerful learning procedures, that can discover good internal representations.

In a network, that uses local representations, it may be feasible to set all the weights by hand because, each weight typically corresponds to a meaningful relationship between entities in the domain. If however, the network uses distributed representations, it may be very hard to program it by hand and so a learning procedure may become essential. Some learning procedure, such as the perceptron learning procedure, are only applicable if the desired states of all the units in the network are specified. Other, more recent learning procedures operate in networks that contain hidden units, whose desired states are not specified by the input or the desired output of the network.

Connectionist learning procedures can be divided into three

broad classes : supervised procedures, which require a teacher to specify the desired output vector, reinforcement procedures, which only requires a single scalar evaluation of the output, and unsupervised procedures, which construct internal models that capture regularities in their input vectors without receiving any additional information [5].

### **Linear associator**

In a linear associator, the state of an output unit is a linear function of the total input that it receives from the input units. A simple Hebbian procedure for storing a new association is to increment each weight  $w_{ij}$ , between the  $i$  th input unit and the  $j$  th output unit by the product of the states of the units.

$$\Delta w_{ij} = u_i u_j \quad (2.1)$$

where  $u_i$  and  $u_j$  are the activations of an input and an output unit. After a set of associations have been stored, the weights encodes the cross correlation matrix between the input and the output vectors. If the input vectors are orthogonal and have length 1, the associative memory will exhibit perfect recall. If the input vectors are not orthogonal, the simple Hebbian procedure is not optimal [5].

### **Nonlinear associative nets**

Non-linear associators perform better than the linear associators in the presence of non orthogonal input vectors. Here the weights all start at 0 and associations are stored by setting a weight to 1 if ever its input and output units are both on in any association. To recall the association, each input unit must have its threshold dynamically set to be just less than  $m$ , the number of active input units.



Hopfield nets store vectors whose components are all +1 or -1, using the simple rule of (2.1). To retrieve a stored vector from a partial description, the network is started at the state of partial description and updates the state of the units repeatedly, one at a time. The iterative retrieval procedure can be viewed as a form of gradient descent in an Energy function :

$$E = - \sum s_i s_j w_{ij} + \sum s_j \theta_j \quad (2.2)$$

where  $s_i$  and  $s_j$  are the states of the two units. Each time an update is done to a unit, it adopts a state that minimizes this energy function. The increase in the change of the global energy caused by changing the unit from state +1 to state -1 is :

$$\Delta E_j = - 2 \theta_j + 2 \sum s_i w_{ij} \quad (2.3)$$

The energy decreases in each step of iteration until the network settles into a local minimum of the global energy function[5].

### Simple supervised learning procedures

Let us consider a network that has only an input and an output layer, with continuous neuron transfer function. A measure of how poorly the network is performing with its current set of input vector is

$$E = \frac{1}{2} \sum ( y_{jc} - d_{jc} )^2 \quad (2.4)$$

where  $y_{jc}$  is the actual state of output for the  $c$  th input output pair, and  $d_{jc}$  is the desired state.

The error measure  $E$  can be minimized starting with any set of weights and repeatedly changing each weight by an amount proportional to  $\frac{\partial E}{\partial w_{ij}}$

$$\Delta w_{ij} = -\epsilon \frac{\partial E}{\partial w_{ij}} \quad (2.5)$$

$$\begin{aligned} \text{where, } \frac{\partial E}{\partial w_{ij}} &= \sum \frac{\partial E}{\partial y_i} \frac{dy_i}{dx_j} \frac{\partial x_j}{\partial w_{ij}} \\ &= \sum (y_j - d_j) \frac{dy_i}{dx_j} y_i \end{aligned} \quad (2.6)$$

( noting that  $x_j = \sum w_{ji} y_i$  )

If the output units are linear, the term  $\frac{dy_i}{dx_j}$  is a constant.

The batch version of this least square procedure sweeps through all sets of inputs accumulating  $\frac{\partial E}{\partial w_{ij}}$  before changing the weights, and so it is guaranteed to move in the direction of the steepest descent. The online version, which requires less memory, updates the weights after each input output cases. This may sometimes increase total error E, but by making the weight changes sufficiently small, the total change in the weights after a complete sweep through all the cases can be made to approximate the steepest descent very closely [5],[6].

#### Perceptron learning algorithm

The perceptron learning technique differ from the least mean square error technique in that, here the derivative in (2.5) is ignored, and only its sign is taken into consideration. So, the weight changes are

$$\Delta w_{ji} = \begin{cases} 0, & \text{if output unit behaves correctly,} \\ +\epsilon, & \text{if output unit should be on} \\ -\epsilon, & \text{if output unit should be off} \end{cases} \quad (2.6)$$

Because it ignores the magnitude of the error, this procedure changes the weights by at least  $\epsilon$ , even when the error is very small.

The major deficiency of both the least squares and the

perceptron learning procedures is that complex mapping between input and output vectors cannot be caught by any combinations of weights in such simple two layer networks [5],[6]. Another deficiency could be that, sometimes the gradient descent may be very slow, because the gradient may be approximately perpendicular to the direction towards the minimum. Another problem is that if the constant  $\epsilon$  in (2.5) large, there may be divergent oscillations, and on the other hand, if it is too small, then the progress could be very slow. A standard method of speeding up in such cases is the Recursive Least Squares technique, which is described in detail in chapter 3.

#### **Backpropagation in multi-layer networks**

The back propagation learning procedure is a generalization of the least squares procedure that works for networks with layers of hidden units between the input and the output units. These multi layer networks can compute much more complicated functions than networks not having hidden layers. But here the learning is much slower because of the presence of the hidden units.

The central idea of backpropagation is that the derivative of (2.5) for the hidden units can be computed efficiently by starting with the output layer and working backwards through the layers. For each input output set the activity levels of each of the units are computed in the forward pass. Then in the backward pass starting at the output layer, the derivative is computed for all the hidden units. For a hidden unit  $j$  in the  $J$ th layer, the only way it can affect the output error is via its effects on unit  $k$  in the  $K$ th layer. So we have,

$$\begin{aligned} \frac{\partial E}{\partial y_j} &= \sum_k \frac{\partial E}{\partial y_k} \frac{dy_k}{dx_k} \frac{\partial x_k}{\partial y_j} \\ &= \sum_k \frac{\partial E}{\partial y_k} \frac{dy_k}{dx_k} w_{kj} \end{aligned} \quad (2.7)$$

So, if the derivative is already known for all units in layer K, it is easy to compute the same quantity for all the units in layer J.

In multi layer networks, the error surface may have many local minima, so it is possible that steepest descent in the weight space will get stuck in a poor local minimum. But if there are sufficiently large number of units and connections then there are typically very large numbers of qualitatively different perfect solutions, and hence the possibility of getting stuck in a poor minimum reduces. In practice, the most serious drawback is the very slow speed of convergence [5].

### **Boltzmann machine**

A Boltzmann machine is a generalization of a Hopfield net, in which, the units update their states according to a stochastic decision rule. The units have states 0 or 1, and the probability that unit j adopts the state 1 is given by

$$p_j = \frac{1}{1 + e^{(-\Delta E/T)}} \quad (2.8)$$

where  $\Delta E = x_j$  is the total input received by the j th unit and T is the 'temperature'. It can be shown that if this rule is applied repeatedly, to the units the network will reach a state of 'thermal equilibrium'. The fastest way to approach low temperature equilibrium is generally, to start at a high temperature and

gradually reduce the temperature. This method is called 'simulated annealing'. It allows the Boltzmann machine to find low energy states with high probability [7].

### **Competitive learning**

Competitive learning is an unsupervised procedure that divides a set of input vectors into a number of disjoint clusters in such a way that, the input vectors within each cluster are all similar to one another. It is called competitive learning because, there is a set of hidden units, which compete with one another to become active. When an input vector is presented to the network, the hidden units which receives the greatest total input wins the competition and turns on with an activity level of 1. All the other hidden units are turned off. The winning unit then adds a small fraction of the current input vector to its weight vector. So, in future, it will receive even more total input from this input vector. To prevent the same hidden unit from being the most active in all cases, it is necessary to impose a constraint on each weight vector that keeps the sum of the weights (or their squares) constant. So, when a hidden unit becomes more sensitive to one input vector, it becomes less sensitive to other input vectors.

### **Reinforcement learning procedures**

A central idea in many reinforcement learning procedure is that, we can assign credit to a local decision by measuring how it correlates with the global reinforcement signal. Various different values are tried for each local variable( such as a state or a weight), and these variations are correlated with variations in the global reinforcement signal. Normally, the local variations

are the result of independent stochastic processes. So, if enough samples are taken, each local variable can average away the noise caused by the variations in the other variables to reveal its own effect on the global reinforcement signal. The network can then perform gradient ascent in the expected reinforcements by altering the probability distribution of the value of each variable in the direction that increases the expected reinforcement. If the probability distributions are altered after each trial, the network performs a stochastic version of gradient ascent [5].

### **Speed of learning**

Most existing connectionist learning procedures are slow, particularly procedures that construct complicated internal representations. One way to speed them up is to use optimization methods such as the Recursive Least Squares that converge faster. If the second derivatives can be computed or estimated, they can be used to pick a direction for the weight change vector, that yields faster convergence than the direction of the steepest descent.

A second method of speeding up learning is to use dedicated hardware for each connection and to map the inter loop operations into analog, instead of digital hardware. The speed of one particular learning procedure can be increased by a factor of about a million if we combine these techniques. This significantly increases the ability to explore the behaviour of relatively small systems. By using the hardware in a different way, might yield a large gain. By dedicating a processor to each of  $N$  connections, a

gain of at most a factor of  $N$  in time at a cost of at least a factor of  $N$  in space can be achieved. For a procedure of complexity  $O(N \log N)$ , a speedup of  $N$  makes a very big difference. For a procedure with time complexity of, say,  $O(N^3)$  alternative technologies and parallelism will help significantly for small systems, but not for large systems [5].

## CHAPTER 3

### THE RECURSIVE LEAST SQUARES ALGORITHM

---



## THE RECURSIVE LEAST SQUARES ALGORITHM

In recent years, considerable attention has been focussed on the development of learning algorithms for use in the Machine Learning Systems (MLS). In this chapter, the Recursive Least Squares algorithm as applied to an MLS consisting of a two layer connectionist network has been described.

Fig 3.1 illustrates the general block representation of an MLS. The model is presented with some training examples with known desired responses and in the training mode, the learning algorithm is used to estimate the model parameters to meet some predefined cost criterion. The learning algorithms, in other words, is used to facilitate robust representation of the training examples in a form, which is usable in the user mode [9].

In the field of communication and signal processing, adaptive algorithms have been used for many years in different areas, such as, channel equalization and modelling, echo cancellation, medical signal processing, and many others. The Recursive Least Squares (RLS) algorithm has been extensively used in these areas [2],[3],[8],[9]. In what follows, the RLS algorithm as applied to an Adaptive Linear Combiner network has been described in detail.

**Adaptive combiner**

Fig. 3.2 illustrates a simple multi input/single output combiner structure. The input vector  $\mathbf{X} = [x_1, x_2, \dots, x_n]^T$ , where the superscript T denotes the matrix transpose, is a set of

features, extracted from the pattern and  $Y$  is the output of the combiner. Given some Knowledge about some problem in the form of input variables and the outputs, it is desirable to estimate the combiner weight vector  $W = [w_1, w_2, \dots, w_n]^T$  in such a way that, when the system is presented with a new set of inputs, it can predict the correct outcome. In other words, the knowledge relating the features to the outcomes is represented as the weight vector in the combiner.

The adaptive combiner structure can be thought of as a multiple input/multiple output single layer connectionist network, whose weights can be estimated using the RLS algorithm. In the following section, the RLS algorithm is described in detail.

#### Analytical derivation of the least squares algorithm

The following notations are used in the derivation:

- $n$  the number of elements in the input vector  $X$ ,
- $l$  the number of elements in the output vector  $Y$ ,
- $m$  the number of training data sets,
- $X(k)$  the  $k$  th training input vector,
- $Y(k)$  the combiner output vector for  $X(k)$ ,
- $D(k)$  the desired output corresponding to  $X(k)$ ,
- $E(k)$  the output error vector corresponding to  $X(k)$ ,
- $W$   $n \times l$  matrix of combiner's weights,
- $R$   $n \times n$  auto correlation matrix of  $X$ ,
- $P$   $n \times l$  cross correlation matrix of  $X$  with  $D$ .

Since,  $Y(k)$  is the response of the combiner to  $X(k)$ ,

$$Y(k) = W^T X(k) \quad (3.1)$$

$$\text{Also, } E(k) = D(k) - Y(k) \quad (3.2)$$



TH-3921

The RLS algorithm aims at minimizing the sum of the squares of these errors over the  $m$  training sets. That is, we have to minimize the quantity  $J_i$ , where,

$$J_i = \sum_{k=1}^m e_i(k)^2, \quad 1 \leq i \leq l \quad (3.3)$$

where  $e_i(k)$  is the  $i$ th element of the vector  $\mathbf{E}(k)$ .

The quantities  $J_i$ ,  $1 \leq i \leq l$  are called performance function. Obviously, it is a function of  $\mathbf{D}(k)$ , and also of  $\mathbf{X}(k)$  and  $\mathbf{W}$ , since they determine  $\mathbf{Y}(k)$ . For a given sequence of vectors  $\{\mathbf{X}(k)\}$  and  $\{\mathbf{Y}(k)\}$ ,  $J_i$  is a function of  $\mathbf{W}$  only. and hence,  $J_i$  are a measure of how well  $\mathbf{W}$  performs to produce an output  $\mathbf{Y}(k)$  which matches the desired response  $\mathbf{D}(k)$ . The choice of  $\mathbf{W}$  that minimizes  $J$  is that value, which has the best performance. This value of  $\mathbf{W}$  is called the optimal value and denoted  $\mathbf{W}^0$ . The best  $J$  can attain is zero, which is attained if  $\mathbf{W}$  can be chosen so that

$$\mathbf{W}^T \mathbf{X}(k) = \mathbf{Y}(k) = \mathbf{D}(k), \quad 1 \leq k \leq m \quad (3.4)$$

If this happens, then each term of the sum (3.3) is zero, and so is the sum itself. There is no worst value of  $J$ , since  $J$  can be made arbitrarily positive with a proper choice of  $\mathbf{W}$ . Also,  $J$  cannot be made negative by any choice of  $\mathbf{W}$ .

Expanding (3.3) we get

$$\begin{aligned} J_i &= \sum_{k=1}^m [d_i(k) - y_i(k)]^2 \\ &= \sum_{k=1}^m [d_i(k)^2 - 2d_i(k)y_i(k) + y_i(k)^2] \end{aligned} \quad (3.5)$$

Using the fact that,  $y_i(k) = \mathbf{W}_i^T \mathbf{X}(k) = \mathbf{X}(k)^T \mathbf{W}_i$ , where  $\mathbf{W}_i$  is the  $i$ th column of matrix  $\mathbf{W}$ , we can write,

$$J_i = \sum_{k=1}^m d_i(k)^2 - 2 \sum_{k=1}^m \mathbf{W}_i^T \mathbf{X}(k) d_i(k) + \sum_{k=1}^m \mathbf{W}_i^T \mathbf{X}(k) \mathbf{X}(k)^T \mathbf{W}_i$$

Let us denote 
$$Q_i = \sum_{k=1}^m d_i(k)^2 \quad (3.6)$$

Also, we see that the auto correlation matrix of  $\mathbf{X}$  is :

$$R = \sum_{k=1}^m \mathbf{X}(k)\mathbf{X}(k)^T \quad (3.7)$$

and the cross correlation matrix of  $\mathbf{X}$  with  $\mathbf{D}$  is :

$$P = \sum_{k=1}^m \mathbf{X}(k)\mathbf{D}(k)^T \quad (3.8)$$

Now,  $J_i$  can be written as :

$$J_i = Q_i - 2W_i^T P_i + W_i^T R W_i, \quad 1 \leq i \leq l \quad (3.9)$$

Where,  $P_i$  represents the  $i$ th column of the matrix  $P$ .

The equations (3.9) can be minimized using results from vector calculus, which states that,  $W_i^0$  is the value of  $W_i$ , which minimizes  $J_i$  if and only if two conditions are satisfied:

$$\nabla J_i \Big|_{W_i=W_i^0} = 0 \quad (3.10)$$

and the Hessian matrix  $H_{W_i}$  is positive definite, where  $\nabla J_i$  is the gradient of  $J_i$  with respect to the elements of  $W_i$ , and  $H_{W_i}$  is the Hessian matrix of  $J_i$  with respect to the elements of  $W_i$ .

Condition (3.10) means that the first derivative of  $J$  with respect to each weight of the vector  $W_i$  must be zero, when evaluated at the optimal weight vector  $W_i^0$ . The Hessian matrix of  $J_i$  is the matrix of the second derivatives. The  $(p,q)$ th element of this matrix is :

$$h_{p,q} = \frac{\partial^2 J_i}{\partial w_{ip} \partial w_{iq}} \quad (3.11)$$

At the optimal point, the hessian matrix must be positive definite, that is, for any nonzero matrix  $V$ , we must have,

$$V^T H_{W_i} V > 0 \quad (3.12)$$

We can now employ the conditions to find the optimal weights. Evaluating the gradient of  $J_i$ , we get,

$$\nabla J_i = \nabla Q_i - 2\nabla(W_i^T P_i) + (W_i^T R W_i) \quad (3.13)$$

Since  $Q_i$  is a constant, its gradient is zero. The gradient of  $W_i^T P_i$  can be found by evaluating each partial derivative:

$$\frac{\partial}{\partial w_{ji}} (W_i^T P_i) = \frac{\partial}{\partial w_{ji}} \left( \sum_{j=1}^n w_{ji} p_{ji} \right) = p_{ji}$$

where,  $p_{ji}$  is the  $(j, i)$  th element of the matrix  $P$ .

$$\text{Thus, we get, } \nabla(W_i^T P_i) = P_i \quad (3.14)$$

Similarly, it can be shown that,

$$\nabla(W_i^T R W_i) = 2R W_i^T \quad (3.15)$$

Combining (3.14) and (3.15) and substituting in (3.13), and noting that,  $\nabla Q_i = 0$ , we get,

$$\nabla J_i = -2P_i + 2R W_i \quad (3.16)$$

For optimality, this gradient must be zero, giving

$$-2P_i + 2R W_i^0 = 0$$

$$\text{or, } R W_i^0 = P_i \quad (3.17)$$

where,  $W_i^0$  means the  $i$  th column of the optimal weight matrix  $W^0$ . The set of  $n$  linear simultaneous equations described by the matrix equation are called the normal equations. Their satisfaction is a requirement for  $W^0$  to be considered the optimal solution.

The second condition for optimality requires that, the hessian matrix be a positive definite matrix. The matrix can be evaluated by evaluating each term:

$$\begin{aligned} h_{pq} &= \frac{\partial^2 J_i}{\partial w_{pi} \partial w_{qi}} \\ &= \frac{\partial^2}{\partial w_{pi} \partial w_{qi}} Q_i - 2 \frac{\partial^2}{\partial w_{pi} \partial w_{qi}} (W_i^T P_i) + \frac{\partial^2}{\partial w_{pi} \partial w_{qi}} (W_i^T R W_i) \end{aligned} \quad (3.18)$$

The first term of the expression is obviously, zero. The first derivative of the second term was found to be  $p_{ji}$  in (3.14), and it is constant. Hence, the second derivative of the second term also vanishes. Differentiation of the third term yields:

$$\begin{aligned}
 h_{pq} &= \frac{\partial^2}{\partial w_{pi} \partial w_{qi}} \left( \sum_{r=1}^n \sum_{s=1}^n w_{ri} r_{rs} w_{si} \right) \\
 &= \frac{\partial}{\partial w_{pi}} \left\{ \sum_{r=1}^n \sum_{s=1}^n \frac{\partial}{\partial w_{qi}} (w_{ri} r_{rs} w_{si}) \right\} \\
 &= \frac{\partial}{\partial w_{pi}} \left( 2 \sum_{s=1}^n r_{sq} w_{qi} \right)
 \end{aligned}$$

$$\text{or, } h_{pq} = 2r_{pq} \quad (3.19)$$

Thus, we see that, each element of the Hessian is twice the corresponding element of the auto correlation matrix R.

$$\text{That is, the Hessian matrix } H_W = 2R \quad (3.20)$$

Thus, the conditions for optimality become :

$$RW^0 = P \quad \text{and } R \text{ is positive definite.} \quad (3.21)$$

Thus, if the matrix R can be inverted, then the normal equations can be used to find  $W^0$ , that is, if  $R^{-1}$  exists, then ,

$$W^0 = R^{-1}P \quad (3.22)$$

From linear algebra, it is known that  $R^{-1}$  exists if R is positive definite. Also, if R is positive definite, the condition (3.21) is satisfied, meaning that, the solution  $W^0$  is unique and can be used to evaluate  $W^0$  according to (3.22). From this analysis, it is seen that, an optimal least squares weight matrix  $W^0$  can be found by using the following steps:

$$(1) \text{ use } x_i(k) \text{ to form } \mathbf{X}(k) \text{ and hence } R = \sum_{k=1}^m \mathbf{X}(k)\mathbf{X}(k)^T$$

$$(2) \text{ find } P = \sum_{k=1}^m \mathbf{X}(k)\mathbf{D}(k)^T$$

(3) if R is positive definite, then find  $\mathbf{W}^0 = \mathbf{R}^{-1}\mathbf{P}$  .

### Recursive least squares

The motivation for developing 'recursive-in-time' algorithms can be seen as follows: if some new training examples are added to the training set, if the above formulas are used, then it will require us to compute the new R and P and evaluate the inverse of R once again. Inversion of a matrix may be a very time wasting process. Hence, we desire to find some procedure by which, the k-th step optimal weight  $\mathbf{W}_k^0$  can be updated to produce  $\mathbf{W}_{k+1}^0$ , the new optimal weight matrix. Such a procedure will build up the optimal weight matrix step by step until the final training set is reached, conserving optimality at each step. The Recursive Least Squares algorithm solves this problem.

**Update** formulas:

The simplest approach to  $\mathbf{W}_k^0$  is the following procedure :

$$(1) \text{ update } R_k \text{ using } R_{k+1} = R_k + \mathbf{X}(k)\mathbf{X}(k)^T \quad (3.23)$$

$$(2) \text{ update } P_k \text{ using } P_{k+1} = P_k + \mathbf{X}(k)\mathbf{D}(k)^T \quad (3.24)$$

$$(3) \text{ compute } R_{k+1}^{-1}$$

$$(4) \text{ compute } \mathbf{W}_{k+1}^0 \text{ using } \mathbf{W}_{k+1}^0 = R_{k+1}^{-1} P_{k+1} \quad (3.25)$$

The auto correlation matrix and the cross correlation matrix are updated and then used to compute  $\mathbf{W}_{k+1}$ . If used directly, the method will be wasteful, because approximately,  $n^3+2n^2+n$  multiplications are required at each step, because of the matrix inversion involved. However, the special

form of the update formula (3.23) can be used to great advantage.

This can be done by using the well known matrix inversion lemma :

$$(A+BCD)^{-1} = A^{-1} - A^{-1}B(DA^{-1}B+C^{-1})^{-1}DA^{-1} \quad (3.26)$$

Substituting  $A = R_k$ ,  $B = \mathbf{X}(k)$ ,  $C = 1$ , and  $D = \mathbf{X}(k)^T$  in relation (3.26) and applying (3.23) yields :

$$\begin{aligned} R_{k+1}^{-1} &= \{ R_k + \mathbf{X}(k)\mathbf{X}(k)^T \}^{-1} \\ &= R_k^{-1} - \frac{R_k^{-1}\mathbf{X}(k)\mathbf{X}(k)^TR_k^{-1}}{1 + \mathbf{X}(k)^TR_k^{-1}\mathbf{X}(k)} \end{aligned} \quad (3.27)$$

From (3.27) it is clear that, given the new input vector,  $R_{k+1}^{-1}$  can be computed directly. There is no need to calculate  $R_{k+1}$ , nor its inversion is necessary. The  $k+1$  th optimal weight matrix is given by:

$$\begin{aligned} W_{k+1}^{\circ} &= R_{k+1}^{-1} P_{k+1} \\ &= \left\{ R_k^{-1} - \frac{R_k^{-1}\mathbf{X}(k)\mathbf{X}(k)^TR_k^{-1}}{1 + \mathbf{X}(k)^TR_k^{-1}\mathbf{X}(k)} \right\} \{ P_k + \mathbf{X}(k)D(k)^T \} \\ &= R_k^{-1}P_k - \frac{R_k^{-1}\mathbf{X}(k)\mathbf{X}(k)^TR_k^{-1}P_k}{1 + \mathbf{X}(k)^TR_k^{-1}\mathbf{X}(k)} + R_k^{-1}\mathbf{X}(k)D(k)^T \\ &\quad - \frac{R_k^{-1}\mathbf{X}(k)\mathbf{X}(k)^TR_k^{-1}P_k\mathbf{X}(k)D(k)^T}{1 + \mathbf{X}(k)^TR_k^{-1}\mathbf{X}(k)} \end{aligned} \quad (3.28)$$

To simplify the expression, let us make the following substitutions :

$$\text{The } k \text{ th optimal vector, } W_k^{\circ} = R_k^{-1}P_k \quad (3.29)$$

$$Z_k = R_k^{-1}\mathbf{X}(k)$$

$$Y(k) = W_k^{\circ T}\mathbf{X}(k)$$

$$\begin{aligned} \text{We get, } W_{k+1}^{\circ} &= W_k^{\circ} - \frac{Z_k Y(k)^T}{1 + \mathbf{X}(k)^T Z_k} + Z_k D(k)^T \\ &\quad - \frac{Z_k \mathbf{X}(k)^T Z_k D(k)^T}{1 + \mathbf{X}(k)^T Z_k} \end{aligned} \quad (3.30)$$



substituting  $q = \mathbf{x}(k)^T \mathbf{z}_k$ , we get,

$$\begin{aligned}
 W_{k+1}^o &= W_k^o - \frac{\mathbf{z}_k \mathbf{Y}(k)^T}{1 + q} + \mathbf{z}_k \mathbf{D}(k)^T - \frac{\mathbf{z}_k q \mathbf{D}(k)^T}{1 + q} \\
 \text{or, } W_{k+1}^T &= W_k^o - \frac{\mathbf{z}_k \mathbf{Y}(k)^T}{1 + q} + \frac{\mathbf{z}_k \mathbf{D}(k)^T}{1 + q} \\
 &= W_k^o + \frac{\mathbf{z}_k [ \mathbf{D}(k)^T - \mathbf{Y}(k)^T ]}{1 + q} \\
 &= W_k^o + \frac{\mathbf{z}_k \mathbf{E}(k)^T}{1 + q} \tag{3.30}
 \end{aligned}$$

where  $\mathbf{E}(k)$  is the error vector for the  $k$  th input  $\mathbf{x}(k)$ .

The update formula for  $R_{k+1}^{-1}$  becomes:

$$R_{k+1}^{-1} = R_k^{-1} - \frac{\mathbf{z}_k \mathbf{z}_k^T}{1 + q} \tag{3.31}$$

This form of RLS has an infinite memory, that is, they can remember all the training data which have been used to train the network. In other words, the final weights are functions of all the sample inputs. This form of RLS is most useful in the presence of stationary input. But if the input data is of non stationary character, that is if they change their character with time, then it is useful to introduce a forgetting factor, thereby diminishing the contribution of the older data. This kind of exponential weighting emphasizes the most recently received data. With exponential weighting, the update formula becomes :

$$W_{k+1}^o = W_k^o + \frac{\mathbf{z}_k \mathbf{E}(k)^T}{\mu + q} \tag{3.32}$$

$$\text{and, } R_{k+1}^{-1} = \frac{1}{\mu} \left\{ R_k^{-1} - \frac{\mathbf{z}_k \mathbf{z}_k^T}{\mu + q} \right\} \tag{3.32}$$

where  $\mu$  is the forgetting factor, and  $0 < \mu < 1$ . But

usually,  $\mu$  is kept within the range of  $0.9 < \mu < 1$ . If  $\mu = 1$  then, the update formula becomes same as (3.30) and (3.31) .

**Computational cost of the RLS algorithm :**

The Recursive version of the Least squares algorithm is computationally less costly than the brute-force evaluation version of the algorithm. The recursive version of the algorithm assumes that the inverse of R and the optimal weight  $W^0$  is already available and then it updates these two matrices, when it receives new training inputs. On the other hand, the brute-force version, whenever it receives new training inputs, constructs the new R and P matrices and inverts R and thereby evaluates new optimal weights  $W^0$ . Since matrix inversion is an  $O(N^3)$  process, it may prove to be extremely wasteful for large size matrices. The recursive version of the algorithm is of  $O(N^2)$  complexity.

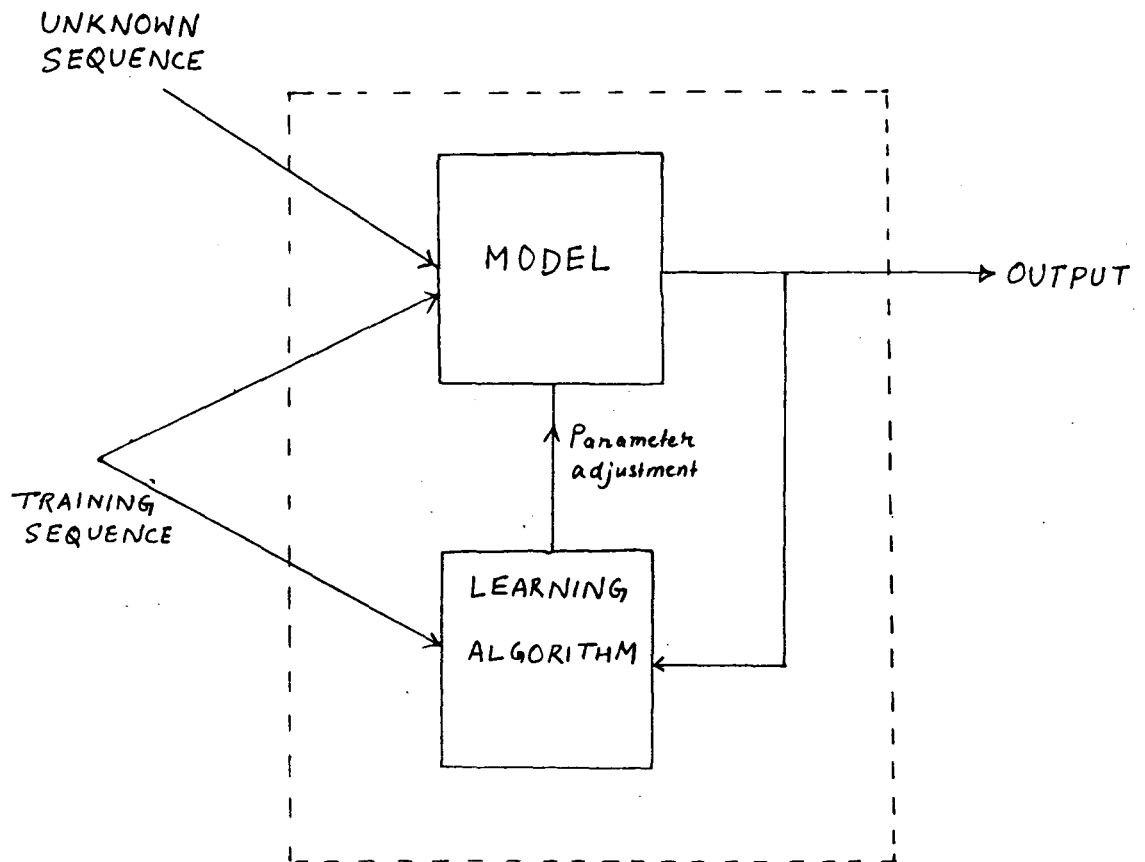


Figure 3.1 General block diagram of an MLS.

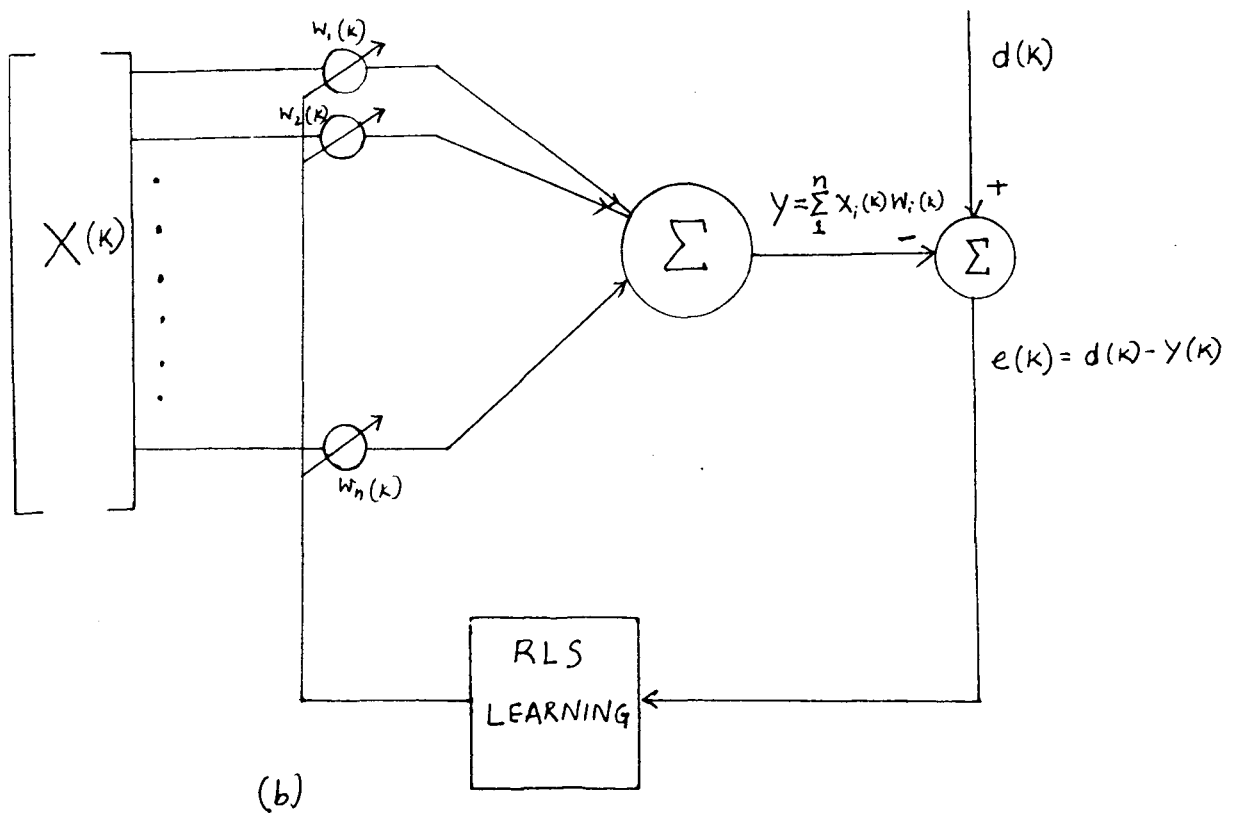
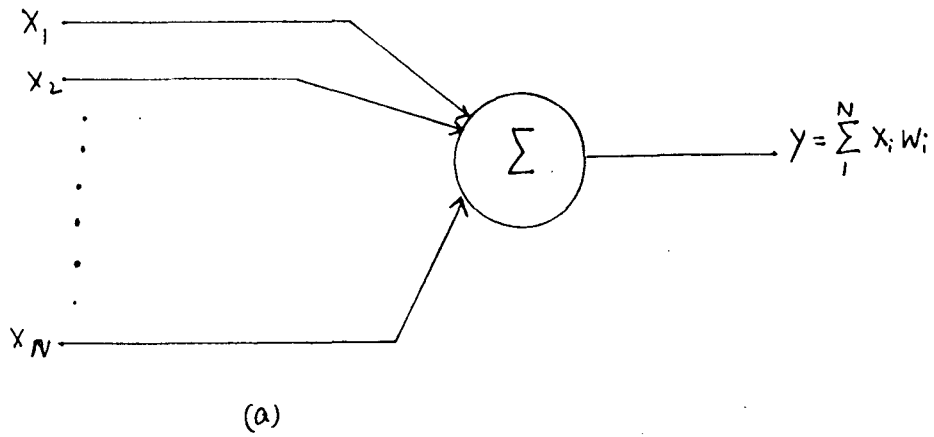


Figure 3.2 Adaptive linear combiner structure.

## CHAPTER 4

# NEURAL NETWORKS AND PATTERN RECOGNITION

---

## CHAPTER 4

### NEURAL NETWORKS AND PATTERN RECOGNITION

Pattern recognition problems require mainly two processes : analysis or feature extraction and classification. The analyzer accepts an input called pattern, which is a very complex physical event, such as optical signals, speech signals or electrical signals etc. The analyzer extracts the main features of the pattern. Its output is the pattern vector. It is a perfectly ordinary vector with no complications. The essential features of such a pattern vector are :

- (a) it has a fixed number of elements called descriptors,
- (b) values of these descriptors in a vector are always known, and are always numeric and
- (c) The order of the descriptors within a pattern vector is always fixed.

The output of the classifier is a series of digital signals, which are mutually exclusive and exhaustive [1].

Machine learning is an important part of pattern recognition. Before the pattern recognition system can operate, the classifier must be taught to behave correctly. The training of the pattern classifier is carried out by Machine Learning Techniques. During the design phase of the classifier, a teacher examines the same pattern as the analyzer and produces a response ( equivalent to the output of the classifier). If the pattern recognition system were perfect, it would produce the same response as the teacher, for all the input patterns.

## Decision surface

A pattern classifier is a device for partitioning of the space defined by the pattern vector  $X$ . That is, it constructs a decision surface in the  $X$ -space, which can be placed in such a way that, it separates the classes defined by the teacher. It should divide the  $X$ -space in some regions, each region containing feature vectors belonging to one and only one class. Supposing this is possible, when this decision surface is placed at an optimal position, both the classifier and the teacher would agree. In this case, we may regard, the teacher as making decisions by use of a decision surface in the  $X$ -space. But in the situation, when the classes are not separable in the  $X$ -space, we cannot draw a decision surface for the teacher. However, we can still draw a decision surface for the classifier.

If the pattern vectors in the  $X$ -space can be separated by using a hyperplane, then the sets are said to be linearly separable. But not all pairs of sets can be separated using such simple surfaces. For example, quadratic surfaces would be required in some situations. In some situations, it is not possible to divide the classes by any finite curve, although any finite samples from these classes can always be separated by a curve of sufficient complexity. Fig 4.1, 4.2 and 4.3 shows the cases of linear decision surface, quadratic decision surface and another quadratic (circular) decision surfaces respectively, in the two dimensional case. Fig 4.4 demonstrates a situation, where it is possible to separate the samples using a complex surface: it is apparent that a simpler (linear) classifier would be more reasonable [1].

### Similarity, distance and compactness

Two patterns produce identical  $\mathbf{X}$  vectors, if the patterns are identical. If two patterns are similar, then we shall usually find that, their corresponding  $\mathbf{X}$  vectors are close together. The distance between two points in a high dimensional space can be defined in many variety of ways. The usual definition of distance in two and three dimensional spaces can be extended to higher dimensional spaces. If  $\mathbf{U}=(u_1, u_2, \dots, u_n)$  and  $\mathbf{V}=(v_1, v_2, \dots, v_n)$  are two points in the  $n$  dimensional space, the Euclidean Distance between them is defined as

$$|| \mathbf{U}, \mathbf{V} || = \sqrt{ \sum_{i=1}^n (u_i - v_i)^2 } \quad (4.1)$$

Let  $\mathbf{U}$  and  $\mathbf{V}$  be two  $n$  dimensional vectors. Any quantity  $D(\mathbf{U}, \mathbf{V})$  qualifies being called a distance function if it satisfies the three conditions :

$$\begin{aligned} D(\mathbf{U}, \mathbf{V}) &= 0 \text{ , if and only if } \mathbf{U} = \mathbf{V} \\ D(\mathbf{U}, \mathbf{V}) &> 0 \text{ if } \mathbf{U} \neq \mathbf{V} \\ D(\mathbf{U}, \mathbf{W}) + D(\mathbf{W}, \mathbf{V}) &\geq D(\mathbf{U}, \mathbf{V}) \end{aligned} \quad (4.2)$$

On the basis of the above properties, the following are distances:

$$D(\mathbf{U}, \mathbf{V}) = \left\{ \sum_{i=1}^n (u_i - v_i)^r \right\}^{1/r} \quad (4.3)$$

$$D(\mathbf{U}, \mathbf{V}) = \sum_{i=1}^n | u_i - v_i | \quad (4.4)$$

$$D(\mathbf{U}, \mathbf{V}) = \text{Max} \{ | u_i - v_i | \}_{i=1..n} \quad (4.5)$$

From the above definitions it is seen that the definition of a distance is quite arbitrary. Similarly it is also an ill defined quantity. We can assume that any two patterns within the same class are similar and they are dissimilar if they belong to



different classes. Then we are likely to find that the distances between the vectors corresponding to patterns from the same class are small. That is, they are likely to be small compared to the distances between patterns from different classes [1].

Another important concept is that of clusters. A cluster corresponds to a peak in probability density function. A class may contain several well separated clusters. A good example of this is that of type written letters of different fonts. While it is unreasonable to expect all the 'A's to be mapped into the same small region of the X-space by the analyzer, we can expect clusters of points corresponding to each font. In such cases, the inter cluster distances are large compared to the intra cluster distances. The Mahalanobis Distance  $r_i$  between the point  $U$  to the  $i$ -th cluster is given by:

$$r_i^2 = (U-M)^T c_i^{-1} (U-M) \quad (4.6)$$

where  $M$  is the centroid of the  $i$ -th cluster, the superscript  $T$  denotes matrix transpose operation, and  $c_i$  the covariance matrix of the  $i$ -th cluster [8].

$$m_i = \frac{1}{N} \sum X \quad (4.7)$$

$$c_i = \frac{1}{N} \sum XX^T - MM^T \quad (4.8)$$

Here  $N$  denotes the number of vectors in the  $i$ -th clusters. The form of the covariance matrix corresponds to an ellipsoidal cluster, where the correlation between the features is expressed by the non-zero terms in the non-diagonal terms in the matrix.

## Neural networks and adaptive pattern recognition

The adaptive threshold element can be used for pattern recognition and as a trainable logic device. It can be trained to classify input patterns into two categories. For these applications the zeroth weight  $w_0$  has a constant input +1, which does not change from input pattern to pattern. Varying this zeroth weight varies the threshold of the quantizer.

### Linear separability

With  $n$  binary inputs and one binary output, a single ADALINE is capable of implementing certain logic functions. There are  $2^n$  possible input patterns. A general logic implementation would be capable of classifying each pattern as either +1 or -1, in accord with the desired response. Thus there are  $2^{2^n}$  possible logic functions connecting  $n$  inputs to a single output. A single neuron is capable of realizing only a small subset of these functions, known as linearly separable logic functions. These are the set of the logic functions that can be obtained with all possible settings of the weight values.

Fig 4.6 shows a two input neuron, and fig 4.7 shows all of its possible binary inputs in the pattern vector space. In this space, the co ordinate axes are the components of the input pattern vector. The neuron separates the input patterns into two categories, depending on the values of the input signal weights and the bias weight. A critical thresholding condition will occur when the analog response  $y$  equals zero:

$$y = w_1x_1 + w_2x_2 + w_0 = 0 \quad (4.9)$$

or,

$$x_2 = -\frac{w_0}{w_2} - \frac{w_1}{w_2} x_1 \quad (4.10)$$

Figure 4.7 graphs the linear relations, which comprises a separating line having

$$\text{slope} = -w_1/w_2 \quad \text{and intercept} = -w_0/w_2 \quad (4.11)$$

The three weights determine the slope, intercept and the side of the separating line that corresponds to a positive output. The opposite side of the separating line corresponds to a negative output, while the line itself is the locus of all the input patterns resulting in a zero analog output.

With two inputs, a single neuron can realize almost all the logic functions. With many inputs, however, only a small fraction of all possible logic functions are linearly separable. The single neuron can realize only linearly separable functions and generally cannot realize most functions. However, combinations of neurons or networks of neurons can be used to realize non linearly separable functions [12].

#### **Non-linear separability**

The input/output mapping obtained in fig (4.7) illustrates a linearly separable function. An example of a nonlinearly separable function with two inputs is the XOR function:

$$\begin{array}{ll} (+1, +1) \rightarrow -1 \\ (+1, -1) \rightarrow +1 \\ (-1, +1) \rightarrow +1 \\ (-1, -1) \rightarrow -1 \end{array} \quad (4.12)$$

No single straight line exists that can achieve this separation of input patterns.

In the network shown in fig 4.8 two ADALINES are connected to an AND logic device to produce an output. Systems of such types are called MADALINES. With weights suitably chosen, they can realize the non-linearly separable function (4.12). The separating

boundary in the pattern space is shown in fig 4.9.

MADALINES are constructed with many more inputs, with many more neurons in the first layer, and with fixed logic device such as AND, OR and MAJority vote takers in the second layer. These three functions are in themselves threshold logic function, as illustrated in fig 4.10. The weights shown will implement these functions, but the weights are not unique [12].

### **Layered neural nets**

The MADALINES of 1960s had adaptive first layers and fixed threshold functions for the output layers. The feed forward neural networks of 1980s have many layers and all layers are adaptive. These networks are more powerful than the MADALINES. Because of the non linear elements present in each neuron, they can pick up the non linearities in the training patterns very efficiently. Because of this, they are very efficient in classification of the non linearly separable functions [12].

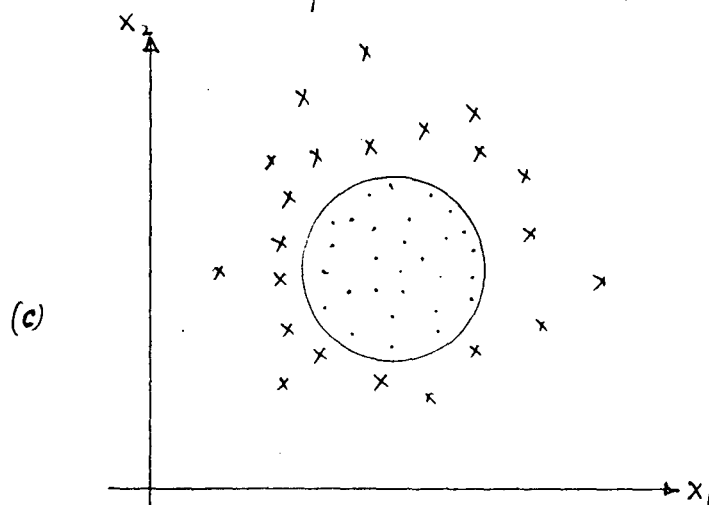
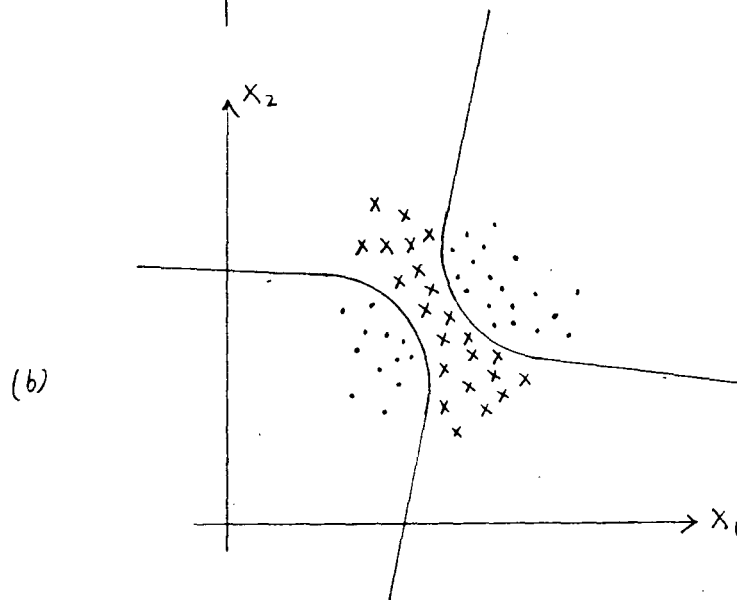
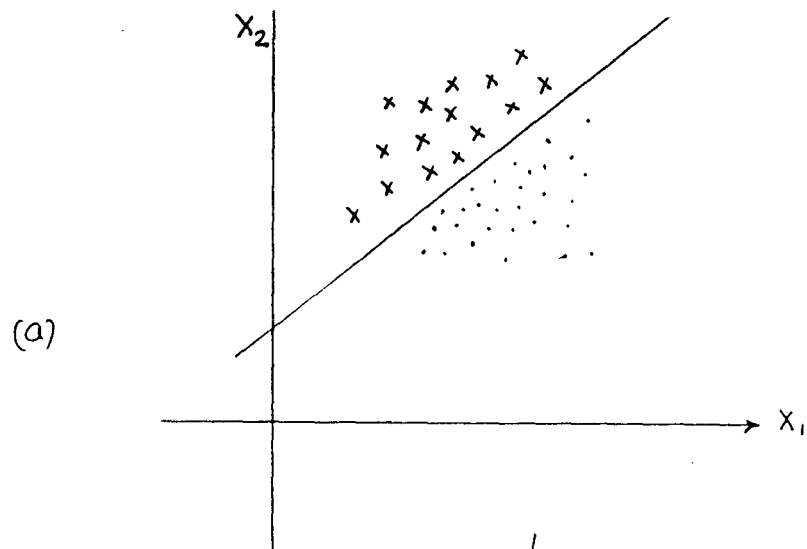
### **Handwritten character classification**

The problem of handwritten character recognition has invoked great research interest for a very long time. There are many difficulties in handwritten character recognition, because of the presence of large degree of variations in the data. Not only are there some changes and distortion of characters from one individual to another, but also, there are some variations from the same individual at different times. Furthermore, difficulties may result from problems such as complexity of characters, similarity of different characters etc [11].

A widely accepted approach is to perform a feature extraction, followed by a classification. The feature classification is

usually predefined and problem dependent. It requires most of the design effort and determines the performance of the whole system to a great extent. The classifier usually incorporates a trainable pattern classifier.

The traditional methods for recognition of handwritten characters have been various statistical techniques, which require a large amount of data, and template matching and graph theoretic approaches, which require quite detailed programming. A more recent and appealing approach is the neural network technique. They can provide a very flexible tool, which allows integration of the feature extractor and the classifier in a single trainable system. As a consequence, the demands of a preprocessor is greatly reduced. Such methods result in high recognition accuracy. Also, neural networks involve only simple arithmetic operations, with a very simple control structure. They can be easily implemented on a digital computer [4].



- (a) Figure 4.1 Linearly separable classes  
 (b) Figure 4.2 A quadratic (Hyperbola) decision surface  
 (c) Figure 4.3 A circular decision surface

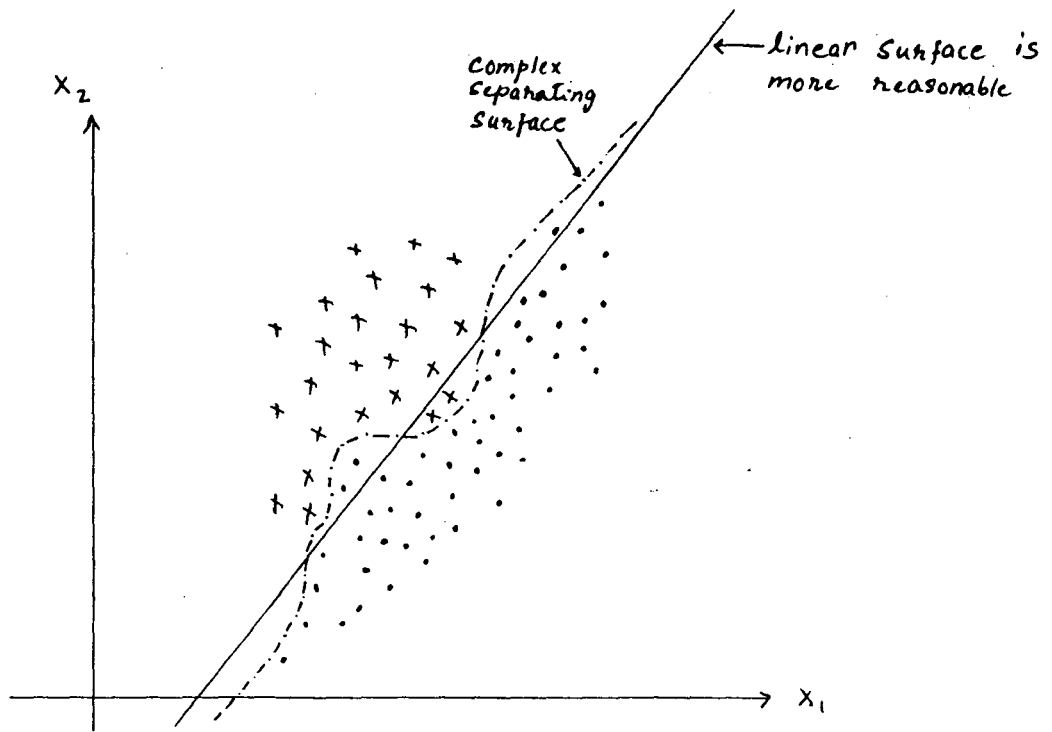


Figure 4.4 Overlapping classes

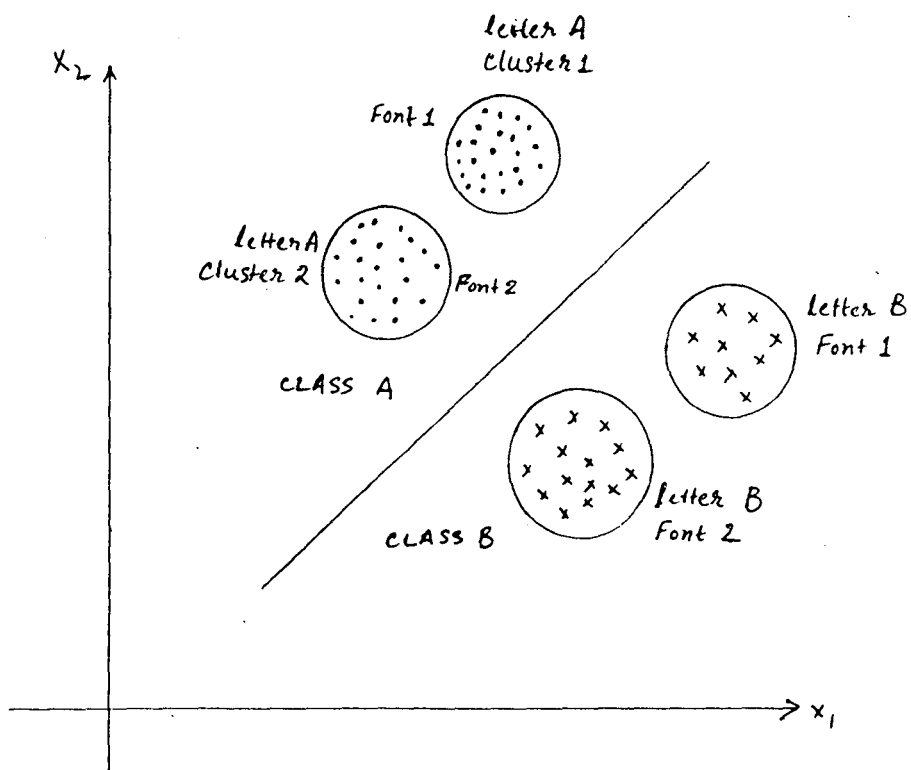
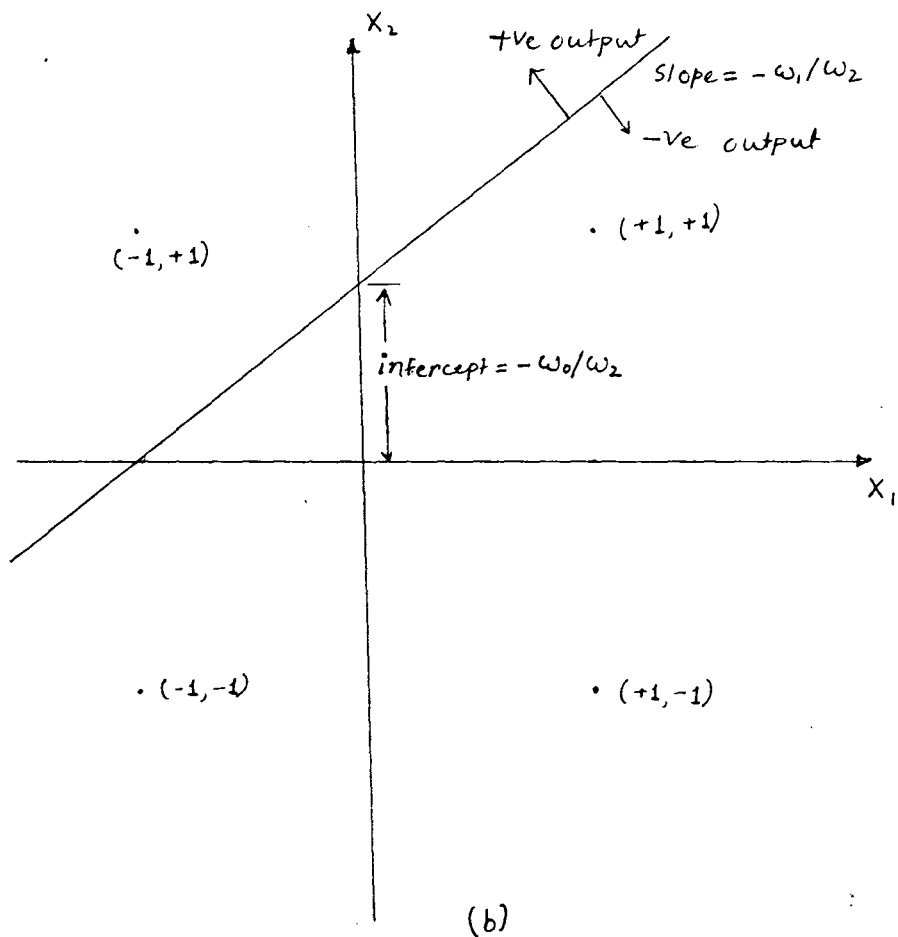
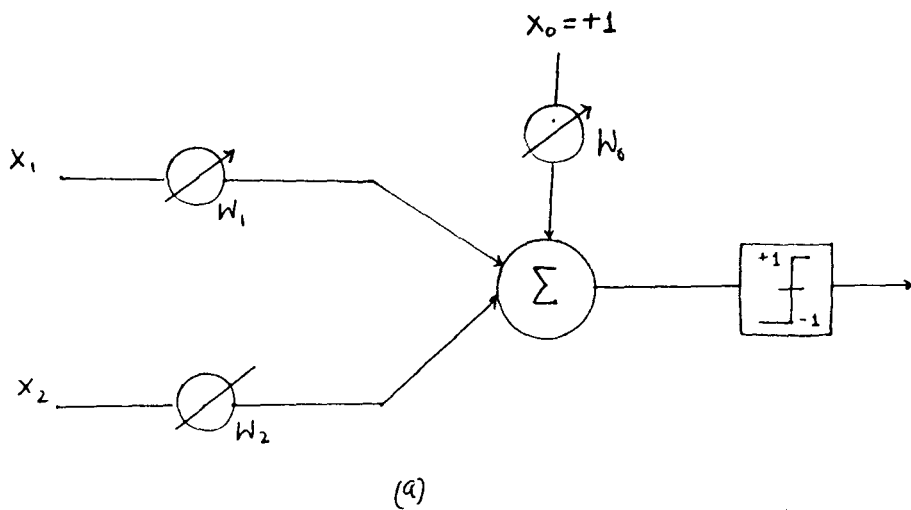


Figure (4.5) An idealized descriptor showing clustering due to different fonts.



(a) Figure 4.6 A two input neuron

(b) Figure 4.7 Separating line in pattern space for a two input neuron.



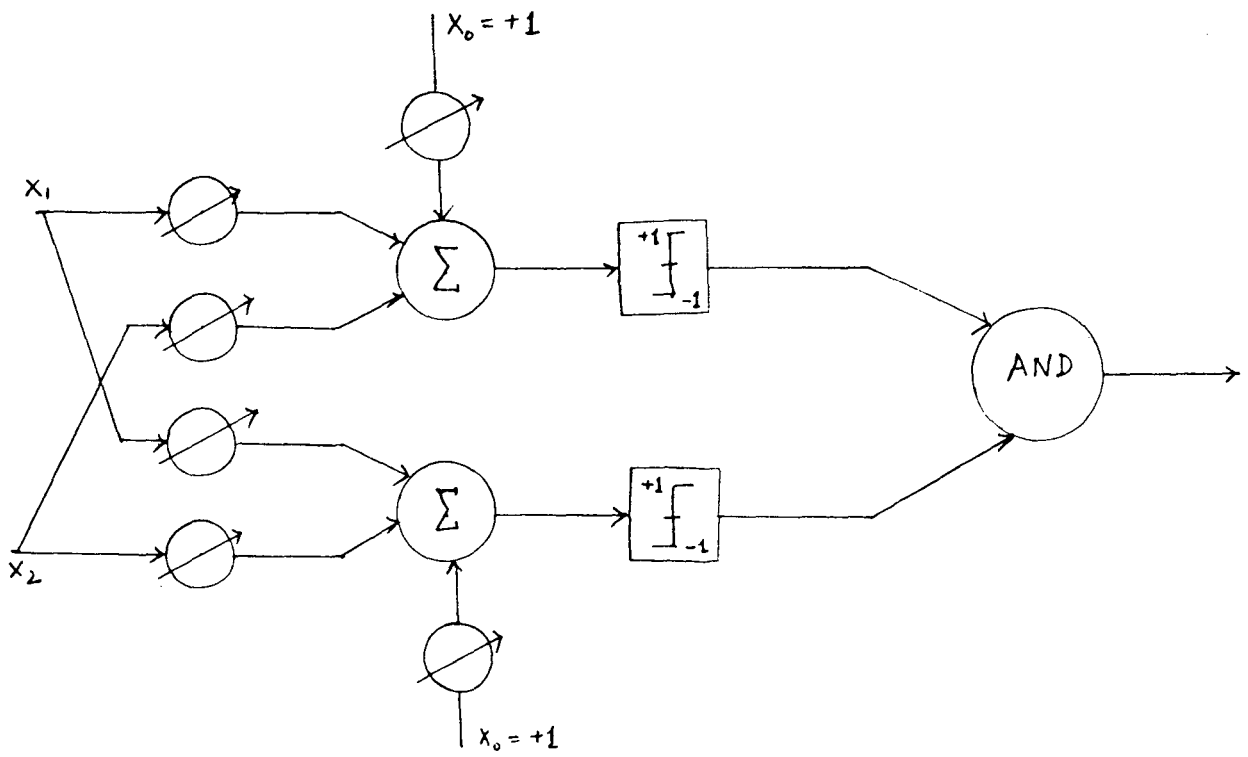


Figure 4.8 A MADALINE

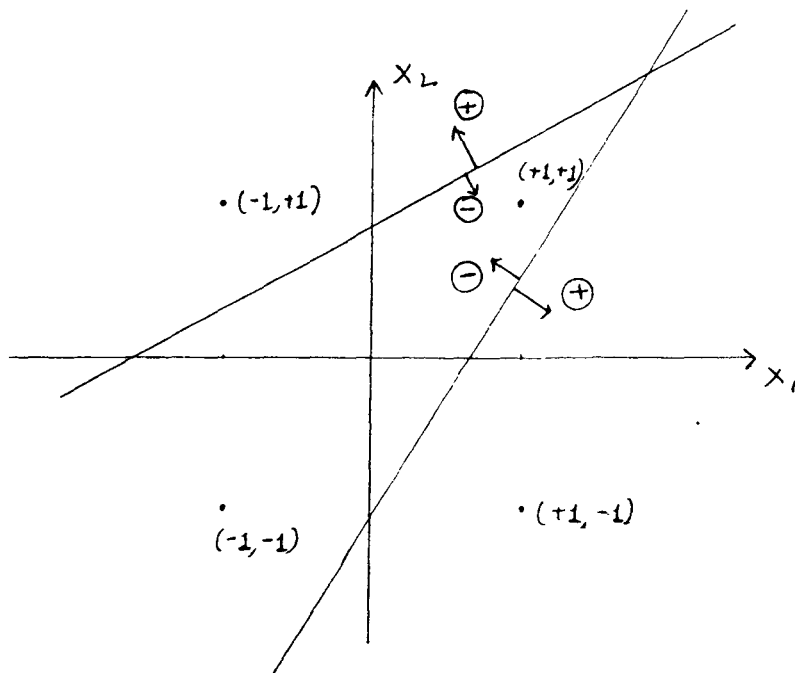


Figure 4.9 An XOR function and separating boundaries for MADALINE of Fig. 4.8

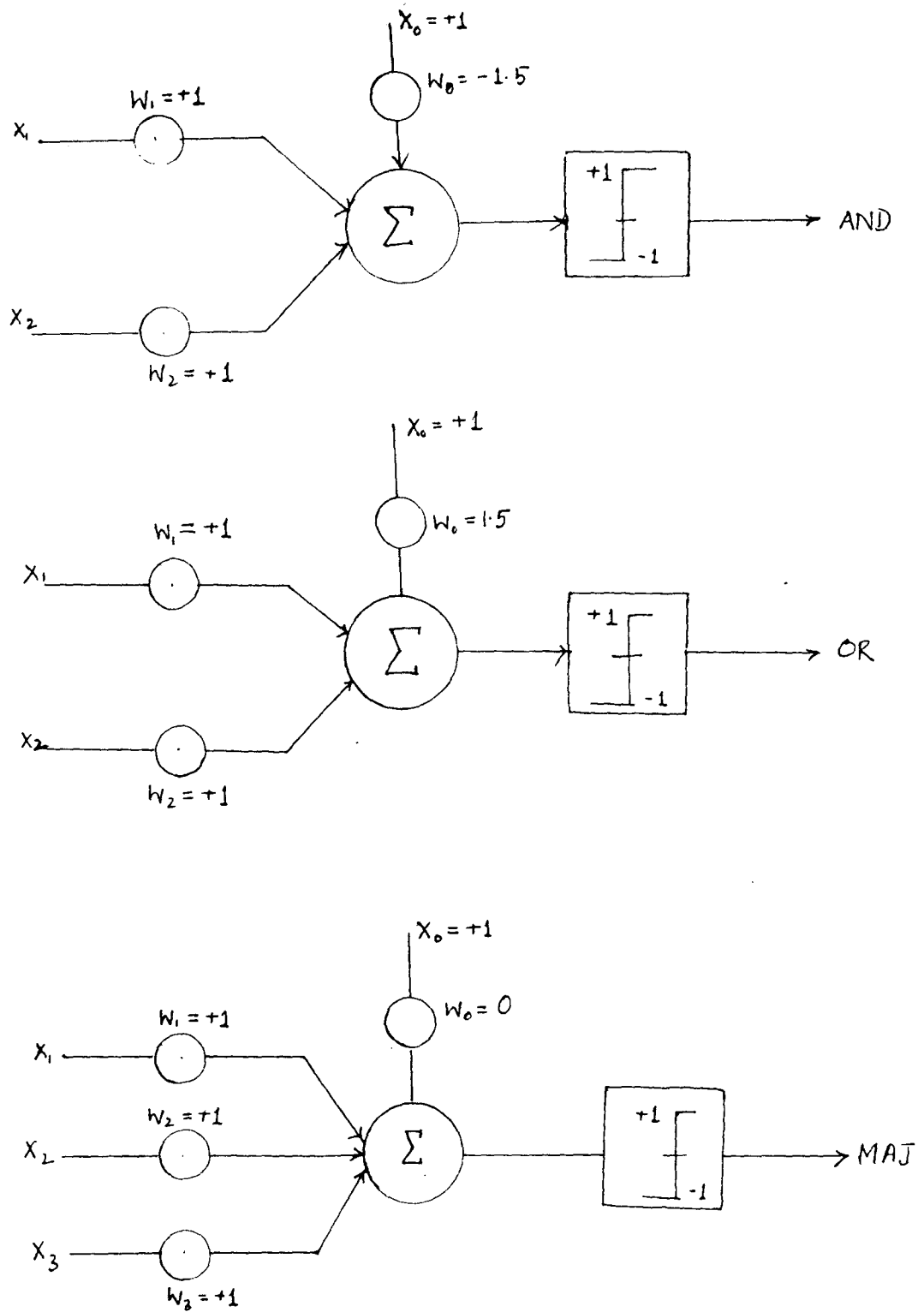


Figure 4.10 Neuronal implementation of AND, OR and MAJORITY Logic functions

## **CHAPTER 5**

### **IMPLEMENTATION OF THE RLS ALGORITHM FOR HANDWRITTEN CHARACTER RECOGNITION**

---

## CHAPTER 5

### IMPLEMENTATION OF THE RLS ALGORITHM FOR HANDWRITTEN CHARACTER RECOGNITION

In this chapter, the implementation of the Recursive Least Squares algorithm for training a two layer connectionist network has been described.

#### Problem specification

The network model : a two layer linear combiner.

The training algorithm used: the RLS algorithm.

The handwritten characters to be identified are 'A', 'B', 'G', 'R', 'S' and 'X', in uppercase.

The work involved developing a Pascal program to simulate a two layer connectionist network ( a linear combiner ) and to implement the Recursive Least Squares algorithm to train the network to recognize handwritten letters. It was decided to teach the network to differentiate the handwritten uppercase letters A, B, G, R, S and X. The input data for training the network were collected from 25 individual. Each of the subjects was given graph papers containing squares of 8 X 8 boxes. Then they were asked to write these six letters, in uppercase, large enough to fill the boxes. After the letters were entered, the subjects were asked to shade in any square of the boxes, which has been intersected by any part of the letters they wrote (fig. 5.1). Out of these 25 sets collected, 21 sets were used as training data to train the network. The remaining 4 sets were used for testing purposes. The

data were entered into a file as a sequence of ones and zeros, representing the filled and unfilled squares of the boxes respectively. Only six letters were to be identified. So, an additional three bit letter identification code was also stored in the text file along with the data which would be used during supervised training of the network.

### **The Network**

The network chosen for this work was a linear combiner network, having an input layer and an output layer. No hidden layer was considered. The network was configured to have 65 input elements : 64 inputs representing the 8 X 8 squares of the digitized letters and an additional biasing input, always set to +1, was used as the 65 th input. Since a minimum of three bits are required for identification of six letters, the network was configured for three output elements.

### **Supervised learning**

The training of the network was done over the 21 sets (126 letters) of samples, which were stored in a file as a sequence of ones and zeros, and along with each letter, an identification code was also stored. The letter identification codes used are as follows:

A : 0 0 1,            B : 0 1 0,            G : 0 1 1  
R : 1 0 0            S : 1 0 1, and        S : 1 1 0.

After reading the inputs and the desired outputs (letter identification codes), the auto correlation and the cross correlation matrices were formed. The inversion of the auto

correlation matrix, which is a quite large matrix (65X65) , turned out to be the most time consuming portion of the whole training process. After calculation of the inverse of the auto correlation matrix, it was stored in a text file. It was then used to calculate the optimal weight matrix for the training sets.

### **Implementation of the algorithm**

The program has been written in three parts : the first program was meant to be run only once, to calculate the auto correlation matrix  $R$ , its inverse and the cross correlation matrix  $P$  for the initial training data. Then it calculates the optimal weight matrix  $W^0$  for the training data. Once they are calculated, the matrices are stored in files for later use by the recursive version of the algorithm. Keeping in view the large size (65X65) of the auto correlation matrix, the inverse was calculated using an iterative method, which calculates a least square approximation of the matrix rather than using direct method such as the Gaussian elimination method or the L-U decomposition method, which are unlikely to provide satisfactory results, even with double precision arithmetic.

The second program implements the recursive version of the least squares algorithm. This program is meant to be used in case new additions were made to teach the network with more training data. The program reads in the  $R^{-1}$  matrix and the optimal weight matrix  $W^0$  from the files where they had been stored by the first program and updates them step by step. After all the training data are dealt with, it stores the new  $R^{-1}$  and  $W^0$  in the same files from where they had been read in. The algorithm can be described

in ten steps as follows :

- (1) Read new input  $\mathbf{X}(k)$  from file
- (2) Read new desired outcomes  $\mathbf{D}(k)$  from file
- (3) Compute the output vector  $\mathbf{Y}(k) = \mathbf{W}_k^{\circ T} \mathbf{X}(k)$
- (4) Compute the error vector  $\mathbf{E}(k) = \mathbf{D}(k) - \mathbf{Y}(k)$
- (5) Compute the vector  $\mathbf{Z}_k = \mathbf{R}_k^{-1} \mathbf{X}(k)$
- (6) Compute  $q = \mathbf{X}(k)^T \mathbf{Z}_k$
- (7) Compute  $v = 1/(1+q)$
- (8) Compute  $\mathbf{Z}_k' = v \mathbf{Z}_k$
- (9) Update optimal weight vector  $\mathbf{W}_k^{\circ}$  to  $\mathbf{W}_{k+1}^{\circ}$  :

$$\mathbf{W}_{k+1}^{\circ} = \mathbf{W}_k^{\circ} + \mathbf{Z}_k' \mathbf{E}(k)^T$$

- (10) Update the inverse correlation matrix  $\mathbf{R}_k^{-1}$  to  $\mathbf{R}_{k+1}^{-1}$  :

$$\mathbf{R}_{k+1}^{-1} = \mathbf{R}_k^{-1} - \mathbf{Z}_k' \mathbf{Z}_k^T$$

The algorithm assumes that the matrix  $\mathbf{R}_k^{-1}$  and the initial optimal weight matrix  $\mathbf{W}_k^{\circ}$  are available initially. So, they are to be read in from the files where they are stored from their earlier calculations.

The third program is used for testing the network. It reads the test data from a file, and reads the stored optimal weight matrix of the network and calculates the output of the network :

$$\mathbf{Y} = \mathbf{W}^T \mathbf{X}$$

Since the output vector will contain real numbers, a simple thresholding (the step function, shifted to the right by 0.5 unit) is done to get a binary output. That is, if  $y_i$ ,  $1 \leq i \leq 3$ , is an analog output of the network then the  $i$ -th digitized output of the network will be +1 if  $y_i \geq 0.5$ , and 0 if  $y_i < 0.5$ .

The software was implemented on a DEC VAX 11/780 using

Pascal.

## Results

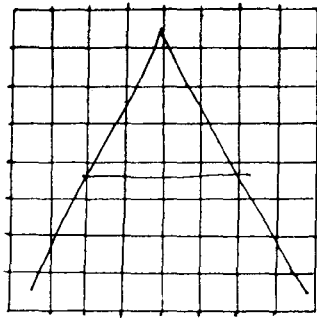
The first program, used to calculate the inverse of the auto correlation matrix, was found to be the most time consuming portion of the training process. The inversion of the 65 X 65 auto correlation matrix took around 8 hours of CPU time on VAX 11/780.

After running the testing program on the training data, it was found that, the program identified 22 characters correctly out of the total 24 characters in the testing data sets.

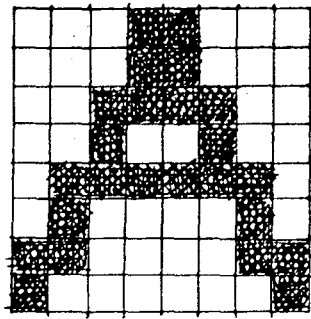
The success rate =  $(22/24) * 100 = 91.7 \%$  .

The letters of the test sets, that the network failed to recognise are shown in fig. 5.2.

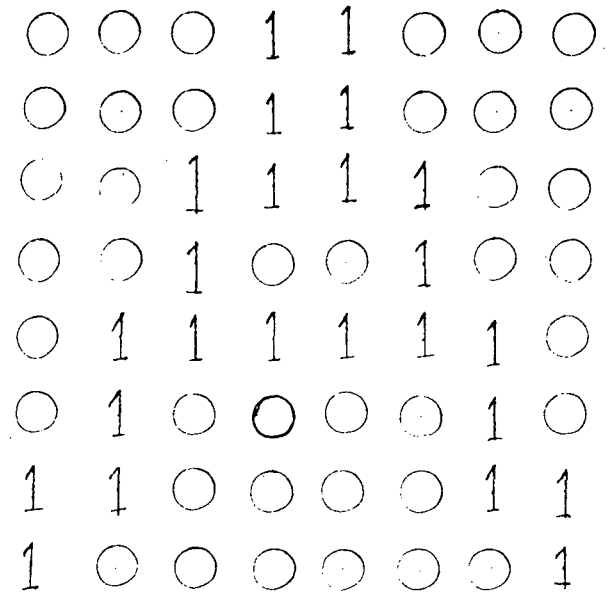




(a)

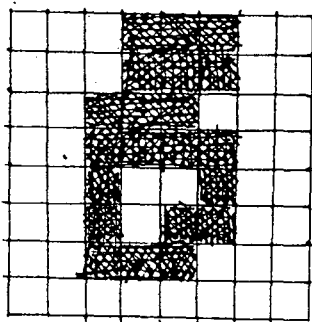


(b)

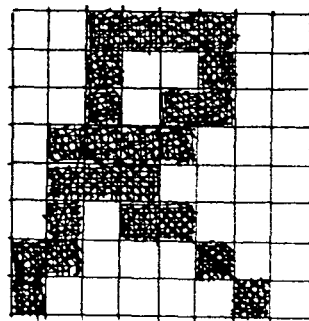


○ ○ 1 ← IDENTIFICATION CODE  
(c)

Figure 5.1 (a) Handwritten 'A'  
(b) 'A' after shading  
(c) File representation of 'A' with identification code.



(a)



(b)

Figure 5.2 The two letters that the network failed to recognize  
(a) a 'B'  
(b) a 'R'

## CONCLUSIONS

Various Neural Network learning procedures have been studied, and in particular, The Recursive Least Squares learning procedure has been studied in detail.

An algorithm for handwritten character recognition based on the Recursive Least Squares learning procedure is developed and an experimental implementation using PASCAL is presented. The simulation results shown by the combiner network were found to be fairly modest. The network is a linear classifier, that can pick up only the linearities in the training data. Since handwritings can vary widely from one individual to another, it may not be possible to separate them using as simple a surface as the hyperplane in the pattern space. A probable method to handle the non-linearities in the training data could be manipulation of the feature set, that is, by using second or third order feature vectors, depending upon the degree of non-linearity.

## BIBLIOGRAPHY

1. Batchellor, Bruce G., Practical approach to pattern classification, Plenum publishing co. ltd., London (1974).
2. Cowan, C. F. N. et al., Adaptive filters, Prentice Hall, Englewood Cliffs, NJ (1985).
3. Crawford, T. M. et al., A machine learning approach to expert systems for fault diagnosis in communication equipments, Computer Aided Engineering Journal, Feb. 1987, pp 31-38.
4. Guyon, I. et al., Design of a neural network character recognizer for a touch terminal, Pattern Recognition, vol. 24, no. 2, 1991, pp 105-119.
5. Hinton, G. E., Connectionist learning procedures, Artificial Intelligence, Vol. 40, 1989, pp 185- 234.
6. Knight, K., Connectionist ideas and algorithms, CACM, Vol. 33, No. 11, 1990, pp 59-74.
7. Lee, B. W. et al., Hardware annealing in analog VLSI neurocomputing, Kluwer Academic Publisher, London, 1991.
8. Mirzai, A. R. et al., Application of recursive least squares algorithm for learning and mathematical reasoning, Engineering Application of AI, Vol. 3, June, 1990, pp 118-126.
9. Mirzai, A. R. et al., Intelligent alignment of waveguide filters using a machine learning approach, IEEE transactions on Microwave theory and techniques, Vol. 37, No. 1, Jan. 1989, pp.186-126.
10. Schweller, K. G. et al., Neural nets and alphabets: introducing students to neural networks, SIGCSE Bulletin, Vol. 1, No. 3, 1989, pp. 2-7.
11. Tappert, C. C. et al., The state of the art in on-line handwriting recognition, IEEE transactions on pattern analysis and machine intelligence, Vol. 12, No. 8, Aug. 1990. pp. 787- 808.
12. Widrow, B. et al., Neural nets for adaptive filtering and adaptive pattern recognition, IEEE Computer, March, 1988, pp.25-39.

